UNIVERSITY OF CALIFORNIA
SANTA CRUZ

LEARNING FROM GAMES FOR GENERATIVE PURPOSES

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Adam J. Summerville

June 2018

The Dissertation of Adam J. Summerville
is approved:

Professor Michael Mateas, Chair

Professor Noah Wardrip-Fruin

Professor Santiago Ontañón

Dean Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Learning from Games for Generative Purposes

by

Adam J. Summerville

Procedural Content Generation has been a part of videogames for most of their existence. One goal has been the generation of content matching a particular design style, and most of the research has been focused on classic AI methods – such as search and constraint satisfaction – to produce content optimizing for some set of constraints and metrics that a researcher sets forth. Recently, there has been a focus on using machine learning to learn the design latent within the content itself, so that a researcher does not need to encode that design knowledge them self; however, this work still includes a large amount of human annotation and guidance.

This dissertation focuses on two systems, *Mappy* and *Kamek*, which work together to learn from a game, so as to be able to generate similar content. *Mappy* is an AI system that uses a wide range of techniques to emulate the process that a human undertakes when they play, observe, and learn from a game. *Kamek* is a machine learning system that learns from game levels and generates new levels of high quality. This dissertation also covers a suite of methodologies for the analysis and presentation of procedurally generated content – with a specific focus on machine learned generators.

I dedicate this to my Mallorie, Clark, and Campbell, who gave me the time, encouragement, and grounding I needed to accomplish this.

Acknowledgments

There are so many people who have helped me get to this point, and I am probably going to leave someone out. I would first like to thank all of the collaborators I've had over these 4 years, who have made my research output seem more impressive than it actually is. I would like to thank Joe Osborn for being my research companion for these last two years, without whom, I would not have the document here. I would like to thank James Ryan and Ben Samuel for including me on the weird journey that is *Bad News*, without which I would never had the chance to talk to 1200 people 1-on-1 in one day. I would like to thank all of the wonderful people in the Game AI community, particularly Mike Cook, Matthew Guzdial, Sam Snodgrass, Julian Togelius, and Santiago Ontañón who have been collaborators, sources of inspiration, and sparring partners at points. I would like to thank my cubicle triangle cohort, Jacob Garbe and Melanie Dickinson, who have put up with my shit for far too long (hopefully, you will no longer think of this as "a hot mess", Melanie). And while we have yet to actually work together yet (cute-vs-good bears not withstanding), I would like to thank Gillian Smith whose research inspired me to come to grad school.

I would like to thank my advisors, Michael Mateas and Noah Wardrip-Fruin. Michael, who always has some tidbit or tangent to go off on about every single AI topic, has helped me hone in on the things that are actually interesting and surprising about my work by saying "Oh, that's interesting, I didn't expect that," which given his breadth of knowledge is usually a sign that you are on to something worthwhile. Noah,

who is usually operating on a couple steps of yomi beyond anyone else, who knows how to ask the question that will make you realize your own misunderstandings or unstated assumptions, has helped me so much in finding how to communicate what I actually want.

I would like to thank my parents, who have always encouraged me to do what I want, even though I didn't always have the courage to actually follow through on that. I would like to thank my children, Clark and Campbell, who, while not often being the best thing for my productivity, have given me the grounding, sanity, and drive to get through these very full four years. You guys light up my day, and I hope one day you will read this and be embarrassed about the words I just wrote, but, hopefully, not the words that are to follow.

Finally, I would like to thank my wife, Mallorie. I know that it was my turn to go to grad school, but I still can never thank you enough for giving me this opportunity. Not only would I not have been able to make it through these years without you, I would never have even taken the chance. You are the one who pushes me to be better, and I hope to one day live up to who you think I am.

Chapter 1

Introduction

Procedural Content Generation (PCG) has been a part of video games for the majority of their existence. *Beneath Apple Manor* [136], the first known game with PCG, was released in 1978, approximately two decades after the first game designed for a computer, *Tennis for Two* [81], and roughly a decade after the first commercial video game, *Periscope* [130]. The proceduralized generation of content for games has largely focused on the creation of "levels" for video games, with the earliest modes allowing for games to contain more levels than would have been possible to store in media at the time. E.g., *Elite* [30] had 256 galaxies of 8 planets each, but was originally intended to have 2^{48} galaxies [6] which given 20 bytes per planet (location, political properties, and name) and 6 bytes per galaxy (location, and orientation) would account for approximately 46.72 petabytes of information, well beyond the 800 kilobyte storage possible on the floppy disks of the time, (or certainly any commonly used physical media used currently). Proceduralizing the generation process allows the galaxies and planets

to be generated at run-time as necessary. PCG as compression of a static experience is one mode of use, but the more common form has been in that of the "infinite" experience. *Beneath Apple Manor* has the player progress through a series of levels, with the goal of a golden apple. The levels are generated anew every time, presenting the player with a seemingly endless set of levels (of course, it is not without as the pseudo-random nature of the generation process will repeat eventually, but this is likely to be beyond a scale that a player will reasonably notice).

In the last decade, PCG has become a focus area for research [174, 175, 213, 167, 75], with a goal being the understanding of different techniques, the properties they afford, and the ability to generate levels with certain properties or style. These approaches typically require the creator of the PCG system to have not only have knowledge of the design space of the artifacts that are being generated, but also a way to elucidate and proceduralize this design knowledge. This is a difficult endeavor for a number of reasons:

1. For a designer in process, design decisions are rarely accurately accessible via introspection [137, 229]

2. Performing a critical deep dive into a piece of work to access the design decisions is not something done lightly, making up entire fields of study (e.g. Digital Humanities)

3. Formalizing all design decisions is difficult, as humans often have hidden biases / foreseeing all possible outcomes of an algorithm is impossible in general (see decid-

ability) and is difficult in specific cases (see the fields of automated testing/code verification)

More recently, PCG research has focused on using machine learning to learn directly from creative artifacts [45, 183, 75, 166], so as to be able to generate content with similar properties. These processes aim to reduce the ability to quantize the design knowledge; however, these processes generally require a fair amount of human intuition, hand-holding produce worthwhile artifacts. So, a research question that needs to be answered is:

Can the process of observation and annotation be proceduralized?

When a human plays a game, they develop theories for how the world of the game works – how things interact, how things move, etc. When a human annotates the level content of a game, they utilize these theories (e.g., "These things are solid" or "That thing hurts the player") to produce an annotation that captures the important mechanical distinctions, while abstracting away things that are not important mechanically. The system presented in Chapter 3 is capable of taking a game and an player control trace as input and producing an annotation of the content found in the game. This requires understanding the processes a human uses during the observation of a game (Chapter 2), so as to operationalize them in a working system.

Given an annotation, the goal becomes the training of a generator that is capable of producing a wide range – and ideally, any possible piece – of content of high quality. Numerous techniques have been applied toward this goal, but it is first

important to understand:

What Theoretical Properties Should a Generator Have?

For a generator to be capable of producing any piece of content, what must it be capable of? Which generation techniques hold these properties, and which fall short? These questions are addressed in Chapter 5.

Given a generator with these theoretical properties, an example system, *Kamek*, capable of generating high quality content is detailed in Chapter 6. The algorithm and data representation are detailed, and a number of experiments are conducted to show how differences in algorithm and data representation affect the resultant generated content.

Given a generator, one wants to understand the properties of the generator. Leading to the question:

What are the properties of the generated content?

Is the generator biased towards some content and away from others? Furthermore, in the domain of machine learned PCG, there is the ultimate question of "Did the generator learn to generate content like the supplied input content?" A set of methodologies for assessing the generative space of a generator are shown in Chapter 11. While it is important to understand the generative space of a generator, it is the actual individual pieces of content that are the ultimate test of a generator. Chapter 12 details methodologies for presenting content in a way that eliminates cherry picking,

while also furthering the understanding of the capabilities of a generator.

1.1 Research Contributions

The primary motivation of this work is the development of systems that are capable of learning from content with minimal human supervision, so as to generate content that has the same qualitative properties. The specific contributions of this work are:

1. An examination the low-level processes that human undertake when they observe and interact with games

2. An operationalization of the processes described in **(1)** in the form of *Mappy*, a system that is capable of observing a game and producing an annotated map for that game

3. A theoretical understanding of the properties that a machine learned procedural content generator must have to be capable of producing any piece of content

4. A general two-dimensional room generator, *Kamek*, and an examination of how different choices in data representation and model parameters affect its generative capabilities

5. A suite of methodologies for the assessment of a procedural content generator for the large-scale analysis of generators, the comparison of multiple generators, and the selection of content

(1) and **(2)** address the motivation of how to learn with minimal human supervision, while **(3)** and **(4)** address the realities of generating content, finally **(5)** addresses how to assess whether a generator (such as the one discussed in **(4)**) has achieved the stated goal.

Part I

Learning From Game

Chapter 2

How Humans Understand Games

Given the goal of learning to generate content with no human annotation, we must first address the problem of human annotation. What is a human doing when they annotate a game level? What gain do we believe is had via this annotation process? And, the core of this work, how can we reasonably proceduralize these processes? [1]

Certainly, at some level, all machine-learned room generators must draw a line in the sand as to how a room will be represented and what portions are annotate and which come from the data (the choice of representation itself being a form of inductive bias). It would be possible to apply image generation techniques [103, 66] to the pixel data of an image representation of a room, and this might produce believable looking images, but it is also possible that these images might have blurry patches, or malformed regions. Moreover, a non-trivial translation process would be required to encode the

[1] Portions of this chapter originally appeared in "Automatic Mapping of NES Games with Mappy" [138], "Charda: Causal hybrid Automata Recovery via Dynamic Analysis" [199], "Automated Game Design Learning" [141], and "What Does That ?-Block Do? Learning Latent Causal Affordances from Mario Play Traces" [193]

image in a way to make it playable. At the other end, it could be possible to directly take the byte array of data stored in the original game that represents a room and learn from that. This would have the nice property of directly learning from the raw representation, but is extremely susceptible to errors in the generation process. Conceivably, being off by a single bit could ruin the entire generation process, whereas something like the image generation process is much less susceptible to a single bit being off (at most changing one pixel of the resultant image).

While both of the aforementioned approaches are conceivable, the vast majority (if not all) of machine learned PCG approaches have had some form of human annotation. Looking at the *Super Mario Bros.* domain, we can compare a number of different approaches. *Super Mario Bros.* contains 122 different "background" tiles (e.g., ground, bushes, pipes, etc.). Some of these background tiles are the same image with a different palette. Some are different images but have identical in-game mechanical properties (e.g., sky and bushes, or the ground and inert bricks). Furthermore, some of these tiles might appear to be identical, but actually contain hidden properties (e.g., a brick might do nothing upon being hit, might produce a single coin, might produce a stream of coins, might produce a mushroom power up, might produce an invincibility star, or might produce a 1-up) (note: these account for an additional 30 tiles). There are also 29 different enemies. Again, some of these are the same image with different palettes, although these different palettes can convey important mechanical distinctions (e.g., green Koopa Troopas will walk off edges, while red Koopa Troopas will turn around upon reaching an edge). This means that a generator that is going to

losslessly encode the information of a *Super Mario Bros.* room will need to be able to produce 181 different tiles. However, a system that operates at this level of fidelity is likely to lose a fair amount of generality. While a ground piece and an inert block are indeed different tiles, they also have identical mechanical function, so it makes sense for generation purposes to encode these properties. The most aggressive annotations come from Shaker and Abou-Zleikha [166] who considered five different types of tile (ground, above ground inert solid, empty, enemy, and other). Snodgrass and Ontañón [182, 183] considered seven different tile types (solid, breakable brick, ?-block, enemy, coin, pipe, and empty). Since Dahlskog et al. [45] used vertical slices of the rooms, their vocabulary when generating was actually composed of up to 73 different slices; however, the slices were composed of 17 different tiles (ground, inert block, breakable brick, ?-block, ?-block containing a powerup, Goomba, Koopa Troopa, winged Koopa-Troopa, Bullet Bill cannon, cannon support, Buzzy Beetle, upper-left pipe, lower-left pipe, upper-right pipe, lower-right pipe, coin, and empty). The least annotated work is that of Guzdial and Riedl [75] who used 102 images as their pattern-matching templates, with the key differences between the 102 and 181 being the ignoring of tiles with hidden information (as this information is not available from video observation) and a coalescing of color information (as their pattern matching occurs in gray scale).

What remains unexamined is why these annotations have been made. We can conject that the work of Guzdial and Riedl is trying to impose a minimal amount of human knowledge, although even the presence of which images show up in their template library and which do not is adding in domain knowledge. As for the others, it

is unexamined as to why the choices in annotation were made, but the fact remains that these annotations were made. All of the approaches intend to learn the design of the original rooms, so as to be able to generate new, playable rooms that are perceptually similar to the originals while not merely regurgitating the original set. At some level, these annotations exist because it is how the authors view the rooms. They have implicitly compressed the rooms, abstracting away information (e.g., sky is the same as a bush is the same as fence, etc.) and compressing the level in a way that is intuitive to them.

This is the process that must be operationalized and proceduralized, if a general learned level generator is to be feasible. However, this raises a number of questions:

- Why is this process important?
- By what guiding principle can we draw a line in the sand as to what needs to be annotated?
- What information needs to be learned and abstracted?

Luckily, the answer to each of these questions stems from one concept, *Operational Logics* (OLs). Operational logics are a framework put forth by Wardrip-Fruin [221], defined by Mateas and Wardrip-Fruin [119], and refined by Osborn, Mateas, and Wardrip-Fruin [142] which relate some process on game state with a communicative strategy. OLs are the atomic pieces that a game is composed of, and while they can be implemented in different manners, they are the smallest unit at which it is important to distinguish.

For instance, the collision, physics, and state logics all multiplex across the same communication channel: an image presented on the screen communicating that an entity is at some location in the world (of a 2 dimensional graphical game). The state portion is composed of the location (typically two coordinates in either $\mathbb{R}^2$ or $\mathbb{I}^2$) and information about the type and state of the entity. This is then communicated to the player as an image at a location on the screen. The specific technical implementation does not matter much, as a textured quad rendered via a 3D renderer or a bitmap image blitted onto the screen buffer both communicate to the player in the same way. This is not to say that technical implementation does not matter, as a given technical implementation will privilege some operations at the expense of others (e.g., rotation and scaling are essentially free for a 3D renderer, while hard to accomplish with bitmap blitting; conversely, pixel perfect alignment are difficult for a 3D renderer but are trivial for a blitting approach); however, for two systems communicating the same information to a player, the underlying technical details are unimportant. Similarly, the coordinate systems could be in $\mathbb{R}^2$ or $\mathbb{I}^2$ or polar coordinates or literally anything. Movement of the entity could be governed by a series of `if` statements, a Runge-Kutta numerically integrated physics simulation, or a series of look-up tables, so long as the actual state holds the same information that is able to be conveyed to the player in the same way.

Given OLs as a framework, how do they address the aforementioned questions:

Why is this process important?

OLs are bundled together pieces of state-process and communication strategy. The act of annotation is a process of converting visual information and game domain knowledge (gathered from play experience) into some bit of what that means. E.g., via play experience a player learns that Mario is able to freely traverse through the sky, bushes, fences in the exact same manner (the Mario image is able to obscure the other images with no apparent change in momentum), and that Mario is unable to freely traverse through the ground, pipes, bricks, and ?-blocks. The player learns that there is a "collision" or "solidity" OL at work for some pairs of game entities and not for others. Given that OLs are the fundamental, atomic grouping of communication and behavior, they are important since they are the entire way the game is understood by the player. The annotation process is important since it is the OL-ification of the game level – the conversion of game fabula (a bunch of pixels presented to the player) into what the games is (a plumber teleporting through pipes to fight turtles). To adequately capture what the game is, we must determine the operational logics at play, which is simply a different lens for the annotation process.

By what guiding principle can we draw a line in the sand as to what needs to be annotated?

As previously mentioned, OLs do not care about the actual underlying technical implementation. As such, we know that we can safely draw a line in the sand that the learned annotations need not learn the actual technical implementation, but rather

can learn a representation that surfaces to the player in the same way. Also, given that an OL is a bundling of communication and state-process, the annotation should only cover things that are both communication and state-process. Given that the goal of an OL is communication to a player, we can reasonably ignore phenomena that are unable to be distinguished by a player, mainly an acceptance of some noise in the process (e.g., whether Mario remains in a state for 14 frames of 15 frames is indistinguishable to a reasonable human player [although, there are, of course, unreasonable human players [4]]). Finally, there exists a catalog of OLs [139], that can act as a guide for which types of state-process/communication bundles are commonly found, guiding us towards a general understanding of the types of OLs we should learn.

What information needs to be learned and abstracted?

The aforementioned catalog provides an outline of the types of OLs one might commonly expect to find (as derived from 47 games [140]) which covers camera logics, chance logics, character-state logics, collision logics, control logics, game-mode logics, linking logics, persistence logics, physics logics, progression logics, recombinatory logics, resource logics, selection logics, spatial pattern-matching logics, and temporal pattern-matching logics.

While any of these could possibly be at play, for the types of games this work is intended to cover, a subset of these will be relevant. To determine which logics will be learned, I will now define the problem space.

For the rest of this work we are going to refer to a *map* as the entirety of the

space that the player will interact within for the game. Maps are made up of a *graph* of linked *rooms*, with the rooms being the direct space that players actually traverse. These rooms contain *entities*, which are governed by dynamical processes (the null, static process is a possible, and indeed common, dynamical process). These entities can interact with each other, through what would commonly be thought of as "collision," and can exhibit different behavior depending on their state. Given observation of a game, we wish to learn which entities are of the same type, the properties of those types (the possible states, the transitions between those states, and the effects of interaction with other entities), their initial locations within a room, and the linking structure between rooms.

To ascertain this information, the relevant logics are:

- Camera Logics – the game typically presents a viewport into the simulated world space. We must deconvolve the camera's position to determine where things are located in the simulation world space.
- Character-State Logics – game entities can have multiple states that govern their properties. We must determine these states, and more importantly the changes between states to determine causal effects of interactions, covered in the next logic.
- Collision Logics – game entities can collide, leading to changes in their state. By frequency, the most common annotations deal with collision logics (i.e.,when entity A and entity B collide, what change is affected to their respective states)
- Control Logics – some game entities can be controlled via some method of input.

Determining which entity is controlled by the player, and when the player retains control, is an important factor for understanding

- Linking Logics – some spaces in a game might be linked together in a way that is not traversable in the same way as the individual spaces are (e.g., two rooms that are linked by a door). We need to determine when the player stops being in one room and traverses to another
- Physics Logics – some game entities' positions are governed by dynamical processes. We need to determine these dynamics to accurately capture the fidelity of a game

In an ideal setting, this process would also incorporate human cultural knowledge, but this work eschews that as beyond scope. Admittedly, this does hinder the communication aspect of the learned OLs as the semiotics can convey mechanical knowledge (e.g., a health granting entity will often contain a red [or green, if the developer understands trademarks [90]] cross on a white rectangle, a reference to the Red Cross organization's mission), but the incorporation of an exhaustive set of human understanding would be well on the way towards the construction of strong AI, which I will leave for future work.

Mappy is a system designed to perform this automatic annotation/OL-ification of a game's map. I will now discuss the sub-problems that *Mappy* faces. In the following chapter, I will discuss *Mappy* in detail, first providing a general overview of *Mappy*, then discussing the input as it is received by *Mappy*, then discussing how *Mappy* addresses

the sub-problems, and finally will discuss future work intended for *Mappy*.

2.1 Human Annotation Tasks

As mentioned before, the goal of *Mappy* is to provide a level that is compressed in a manner compatible with human understanding, as losslessly as possible. This is not to say that *Mappy* annotates as a human would but rather to say that Mappy's annotation is human understandable and parseable. Looking to the human annotation efforts, we see concepts like "solidity" and "inert." *Mappy* should be capable of learning the mechanical properties underpinning these, and while these are relatively simple concepts, a large number of elided human processes are required to get to such concepts. While we do not have to perform object detection and image segmentation, a whole host of low-level perceptual concerns need to be covered for *Mappy* to have a chance of learning useful properties. The following sections 2.2,2.3,2.4,2.5, and 2.6 will discuss these problems. The approaches taken in *Mappy* to resolve them are discussed in Chapter 3.

2.2 Entity Persistence

When a human views a screen, a number of phenomena are occurring. The screen is updating at some rate (89 Hz for a cathode ray tube, 60 Hz for most liquid-crystal displays, 144 Hz for modern organic light-emitting diode screens), displaying frames that persist for fractions of a second. While these images persist for mere frac-

tions of second, they appear continuous via the psychophysical process of flicker-fusion [43]. Furthermore, when these static images have objects translate to nearby positions, humans perceive that these objects are moving via the psychophysical process of apparent motion [222]. Figure 3.3 on 44 shows an example of this apparent motion in *Super Mario Bros.*. As mentioned this relies on the common OL communication channel of an image on the screen denoting an entity existing in the game world.

2.3 Camera Motion

What is meant by camera motion? What is a camera, in the context of a two-dimensional graphical game? Early games were single-screen, meaning they presented a single, static screen that showed the totality of the space that a player would traverse through. For some games (e.g., *Pong* or *Asteroids*) this screen would encompass all of the space in the game, and while entities could enter or exit the screen, that one screen is all the player would ever encounter. Later games would introduce "levels" (which are referred to as *rooms* in this work) [131] wherein the space presented to the player would change, while still keeping the player confined to one screen at a time (e.g., *Pac-Man*, *Galaxian*, *Donkey Kong*). It was not until 1974 that *Speed Race* [207] would introduce a room that scrolled, allowing a player to traverse, in a seamless, continuous manner, a larger space than could be represented on a single screen. This scrolling was often limited to a single axis at a time, with top-scrolling being popular for games involving the piloting of an air or space craft (e.g., *Xevious* or *1942*) and side-scrolling being

popular for similar games (e.g., *Bomber* or *Defender*) and platformer games (e.g., *Super Mario Bros.* or *Jump Bug*) as well. While the cameras present in these games do not reproduce the actual mechanics of a camera (whereas modern three dimensional games do capture some given their handling of field of view [15]), we, as humans, are able to intuit their mechanics as being representative of an actual camera. The question now being, how do we as humans understand that the motion being presented on the screen is not caused by the entire scene moving (no matter if that is what is actually happening in the game's code) but rather that there is some static scenery that the camera is panning over?

A number of different factors are crucial to understanding and proceduralizing this phenomenon. In part, semantic representation knowledge is a large enabler. Properties of entities such as whether they are animate or inanimate allows us to reasonably infer which entities in a scene should be moving (e.g., a human, an animal, or an anthropomorphic object) and which should not (e.g., a tree, a bush, a rock, a brick). This semantic representation knowledge might be noisy (e.g., should a cloud be stationary? On a still day, yes. On a windy day, no) or might lead us astray (e.g., the normally inanimate mushroom is locomotive in *Super Mario Bros.*). Figure 2.1 shows a scene from *Super Mario Bros.* with typically inanimate entities in gray while entities deemed mobile via intuition are in color. Techniques built off this semantic representation knowledge is beyond the scope of this work, and *Mappy* treats all images as unique identifiers, with no understanding of what the images represent.

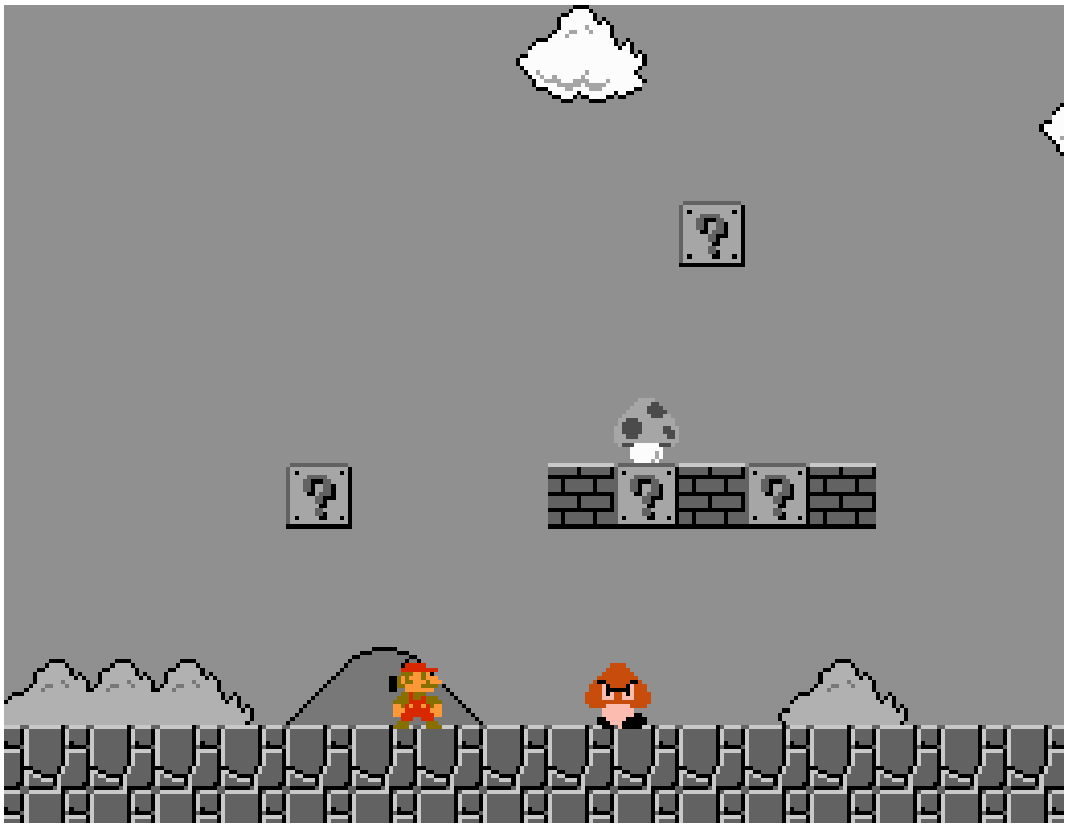


Figure 2.1: A scene from *Super Mario Bros.* where semantic representation knowledge would classify entities that are animate in color and the inanimate entities as gray-scale. The human is obviously animate, and the anthropomorphic mushroom with eyes and feet would reasonably be intuited to be mobile. Note that this intuition is incorrect as the mushroom above the ?-block has the same movement profile as the Goomba.

2.4 Room Detection

As mentioned in section 2.3 motion, games have long had a history of being segmented in a way where the player can freely traverse through a space before discretely transitioning in a manner to a different space. The earliest games with this structure allowed the player to only traverse this larger structure, from here on referred to as the *map*, in one direction, with the method for traversing between rooms being tied to some criterion of "beating" the current room (e.g., reaching Donkey Kong in *Donkey Kong* or consuming all of the pellets in *Pac-Man*). Later games would allow for the player to traverse the *map* in a non-linear manner (e.g., *Adventure* or *The Legend of Zelda*) or quasi-linear manner (e.g., *Super Mario Bros.* does not allow the player to back-track but can choose a number of different paths). These rooms might be connected in a way that honors standard Euclidean geometry (e.g., *Metroid*) or might not (e.g., in *Adventure* it is possible to traverse $\Rightarrow, \Uparrow$ from a location and wind up in a different location than if you traversed $\Uparrow, \Rightarrow$). It is very common for media written about games to refer to specific rooms [22, 227] and it is common for authored maps of games to divide the map into rooms or collections of rooms [16, 10].

Smith and Whitehead refer to *cells*, non-overlapping regions of space that the player can continuously traverse through, and *portals*, the means by which the player can move between different cells, [178] which are analogous to rooms and *doors*, the term which will be used from here on to refer to the connection between rooms. They make this distinction as "Knowing where the cells and portals are in a level helps us

analyze their structure and catalog the many paths through a level." Unsaid in this is that the mega-structure of a game consists of separate rooms (what they refer to as levels) and knowing that structure is important. Without knowing this structure

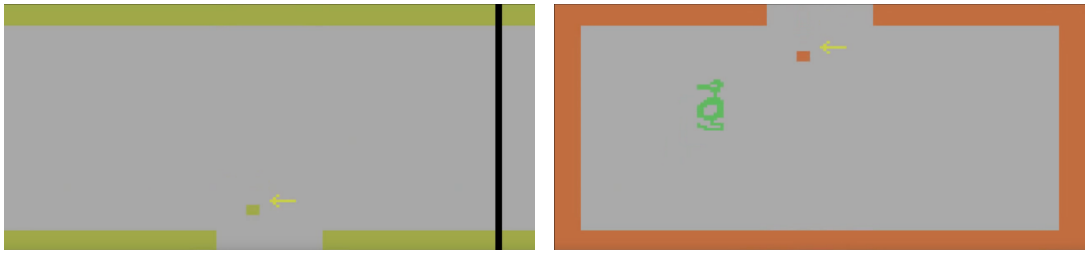
- The wrong information might be extracted (e.g., in *Adventure* recording the map as one single room would necessitate overwriting information due to the non-Euclidean nature)
- Transitions that do not contain spatial information would be hard to reconcile (e.g., when going down stairs or in to a cave in *The Legend of Zelda*, where is that space located in relation to the previous, Euclidean space)
- Important semantic distinctions between rooms would be lost (e.g., in *Super Mario Bros.* an above ground room is different from a water room is different from a dungeon room, etc.)

Operational Logics provides the framework of a *Linking Logic*, the method and communication of the player traversing between connected but distinct spaces. To determine how to ascertain the super-structure of a level vis- 'a-vis the topology of rooms, it is necessary to discuss how the distinction between rooms and their connections are communicated to players.

There are a number of ways of communicating the movement between two connected spaces in a game, but there are two that are most common, *teleporation* and *traversal*.

2.4.1 Teleportation

Teleportation refers to placing the player in an environment that is noticeably different in an instant and represents the most common form of room traversal. There are multiple variants of how this is presented to the player.



(a) (b)

Figure 2.2: Two rooms in *Adventure* that are traversed between. The player exits through the bottom of the first room and winds up in the top of the second.

The simplest is changing between the two rooms in the course of a single frame. Figure 2.2 shows an example of this in the game *Adventure*. When the player moves between rooms, they are in one room on one frame and in the next on the subsequent frame.

However, it is common for games to include some manner of interstitial denoting the traversal. For instance, *Super Mario Bros.* has a number of different interstitials depending on the types of rooms being traversed. Figure 2.3 shows the interstitial presented to players at the start of a world (a set of rooms). This interstitial is presented to the player for a number of seconds, operating as a sort of palate cleanser, allowing the player to see their progress through the game and their current status (number of lives left, amount of points and coins collected, etc.). Within a world, when the player tra-

Figure 2.3: The interstitial shown in *Super Mario Bros.* before the player starts a "world" (a set of rooms).

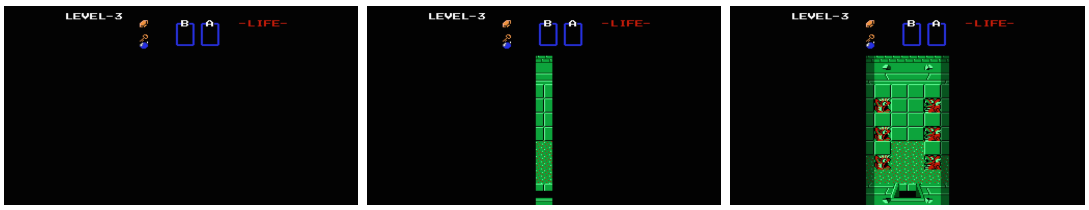


Figure 2.4: Three frames from the interstitial sequence in *The Legend of Zelda* that occurs when the player enters a dungeon.

verses a door between rooms three screens of solid color are shown, first the background color of the previous room is shown for 2 frames (e.g., solid blue if the background color was previously the sky), followed by a black screen for 7 frames, followed by the background color of the new room for 2 frames. This process lasts barely over a tenth of a second, but the brief flash primes the player that a transition between rooms is occurring. When the player enters a "dungeon" in *The Legend of Zelda* a somewhat complex interstitial scene occurs, shown in figure 2.4. First, a solid black screen is shown for 12 frames ($\frac{1}{5}$ of a second), then the UI elements appear, but the screen is still predominantly solid black. Then the black recedes to the left and right sides, as if curtains are being drawn. Once the black has fully receded, the player regains control.

While these all represent different interstitial techniques, ranging from the non-existent instantaneous interstitial to scene setting animations, they all involve drastic changes in what is presented to the player. This allows the intuition that drastic changes in scenery, typically accompanied by loss of player control, and discontinuous movement of the player entity represent a teleportation transition between rooms. However, there are transitions between rooms that do not have such abrupt, drastic changes in what is shown to the player, so it is important to determine the communication strategies for *traversal* doors.

2.4.2 Traversal

While teleportation transitions are very common, it is also very common for transitions to occur in which the direct geometric relationship between two rooms can

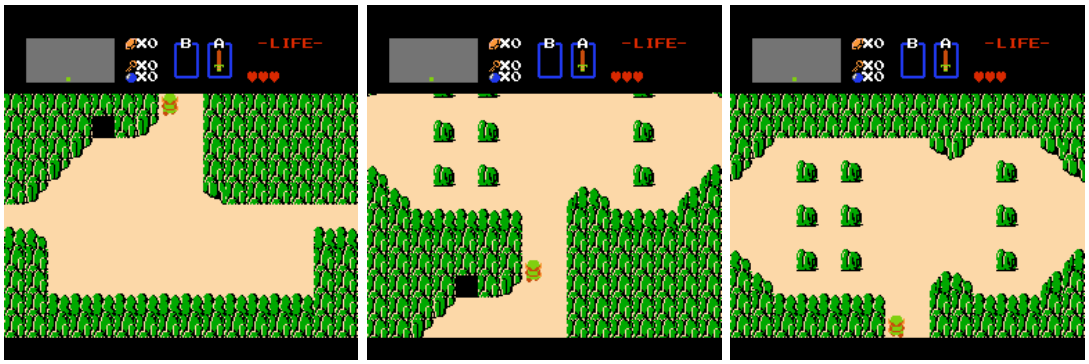


Figure 2.5: Three frames from the traversal interstitial sequence in *The Legend of Zelda* that occurs when the player traverses between rooms connected in a Euclidean manner.

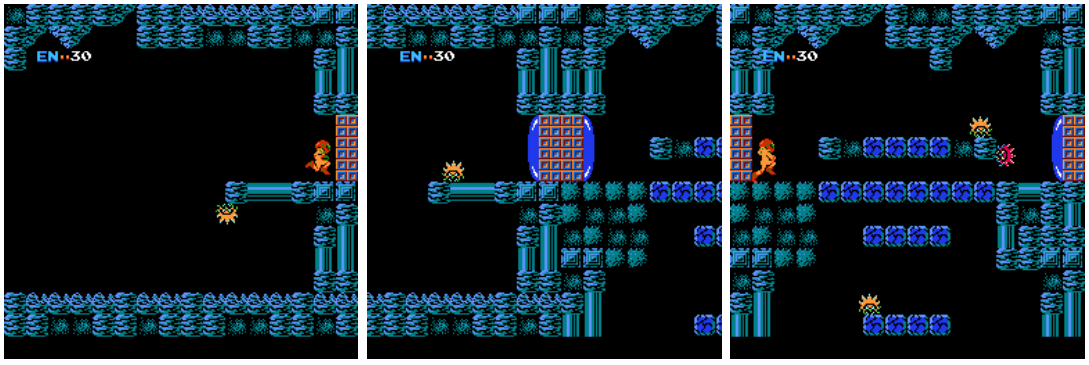


Figure 2.6: Three frames from the traversal interstitial sequence in *Metroid*.

be understood, which here are termed *traversal* doors. For instance, the vast majority of transitions in *The Legend of Zelda* occur between two rooms which are understood to be next to each other in a cardinal direction. Figure 2.5 shows three frames from such a transition. When the player collides with the outer border of a room, a brief interstitial scene is shown where the screen scrolls in the direction that the player appears to be moving (i.e., it scrolls up if the player collided with the top of the screen) while the player's walking animation plays. After fully scrolling in to the new room, the player regains control and continue to move freely. A very similar scene occurs in *Metroid* when the player opens a door, walks through the threshold, the screen scrolls along the direction of the door, and finally the player regains control in the new room, shown in figure 2.6.

From a purely visual standpoint, there is no way to discern that the player is traversing between rooms, as opposed to within a room. Certainly, this could be seen as an eccentric form of the position locking camera behavior as defined by Itay Keren [101], e.g. the player is given a window that they can move freely in, but when coming to the edge of the camera's field, the camera scrolls to a new position. This would be plausible, with no knowledge about what the player is doing, but a player certainly has some degree to which they understand their actions and interactions with the game, so this would be unsatisfactory.

A key aspect of these transitions is the loss of player control. If a room is a place in which the player can freely move about, a key aspect of communicating that something *different* is occurring during the transition to a new room is the removal of

that control. By removing control, while still keeping motion, the player infers that they are moving to a new place, in a way that is distinct from their previous movements. This motion can be of a different modality than their previous motion (e.g., in *Super Mario Bros.* the key way a player transitions to a new room is via the entering of warp pipes, a very different action than their previous running and jumping) or be similar (e.g., in *The Legend of Zelda* the player's standard walking animation occurs during the transition, although the player's character remains still until it is "pushed" by the edge of the screen at the end of the transition), but the key aspect is the loss of control whereby the player is informally informed that they are now in an interstitial space between the rooms.

This raises the question, "How do we know when the player does or does not have control?" From observation of a video, there would be no way to discern when the player has control, as there is no information about the control input to discern what the player is controlling; however, given the instrumented emulator, it is simple to discern player control. To wit, player control is determined when a change in input results in a change in state. Now, it is possible that a change in input results in a change in state that is not communicated to the player (e.g., the initial random seed in *Final Fantasy* is determined via the time since console power on when the player first presses start – potentially leading to a large change in a player's experience); however, given that the goal is to determine when a reasonable player could be expected to notice a difference in their control, it is relatively safe to ignore these "invisible" state changes.

To determine whether the player has control, on each frame all possible inputs

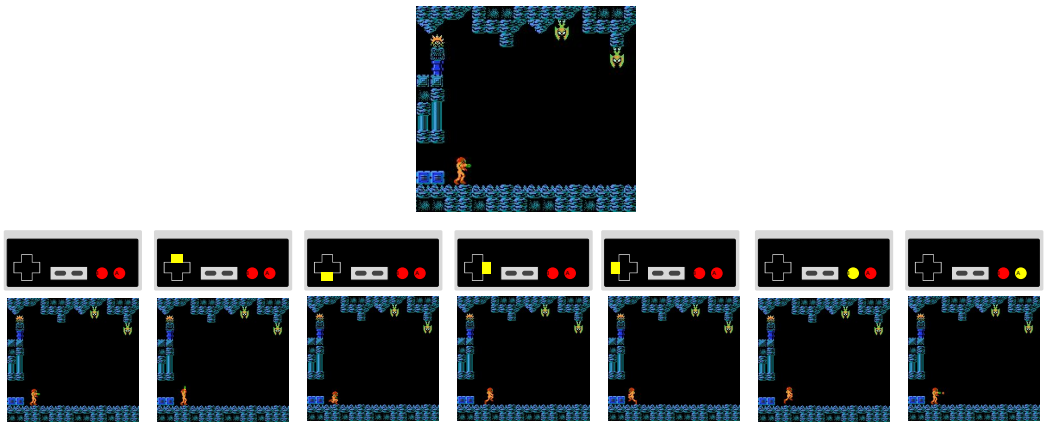


Figure 2.7: The original frame is shown on the top, and the results of each input on the resultant frame are shown on the bottom.

are tried (pressing up, down, left, right on the direction pad, the A button, and the B button). Then progress the emulator through the next n frames with the input as specified by the input play trace. If the any of these inputs results in a difference in what is presented to the player, then it is deemed that the player has control. The lag of n frames is a concession to the fact that not all games have instantaneously observable control (or might not at all points in time). Figure 2.7 shows an example in *Metroid* for determining that the player has control.

To summarize the two types of transitions, they can either be the result of a drastic change in the scene presented to the player, or indicated via the loss of control during a period where the background is still moving (as there are other reasons why the player might lose control).

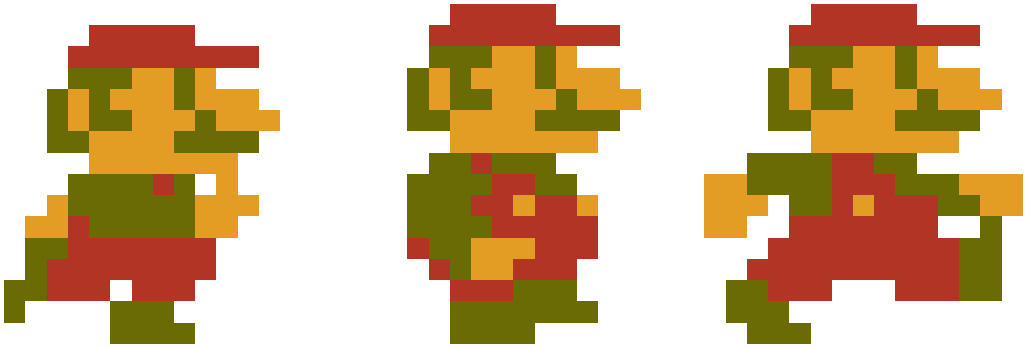


Figure 2.8: The three images that make up Mario's run cycle in *Super Mario Bros.*

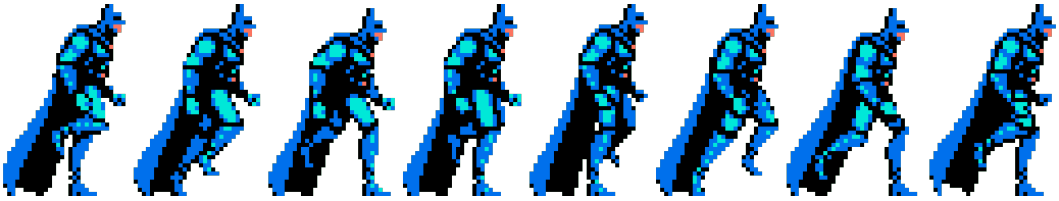


Figure 2.9: The eight images that make up Batman's run cycle in *Batman: The Return of the Joker*

Figure 2.10: A phénakistiscope with a man pumping water from Mclean's Optical Illusions [122]. Reprinted with permission[151].

Figure 2.11: The two halves of a thaumatrope from 1825 which would display a vase full of flowers when used. Reprinted with permission [63].

Figure 2.12: A reproduction of a Victorian zoetrope. Reprinted with permission [54].

2.5 Animation Cycles

Animation, as an artistic practice, has a storied history of roughly two centuries at this point [109]. Starting from the thaumatrope, a toy disc with two images on alternate sides that could be rapidly swapped between (see figure 2.11), (relying on properties of vision similar to those mentioned in section 2.2), through the phénakistiscope, a disc with a sequence of images proceeding around the outer perimeter that would be rotated to create the appearance of motion (see figure 2.10), to the zoetrope, similar to the phénakistiscope except with the images arranged in a cylinder (see figure 2.12), the earliest animations were composed of a small number of images (between two and sixteen), a trait held by most of the games present on the NES. For instance, Mario's run animation in *Super Mario Bros.* has three frames of animation (see figure 2.8) and Batman's run animation in *Batman:Return of the Joker* has eight frames (see figure

2.9) . While these animations are easy to discern given common sense knowledge about typical actions and what they look like, this common sense knowledge is beyond the scope of this work. Pose recognition and estimation [230] is a subset of research in the machine vision community focused on detecting human forms from video or, in some cases [171], a single image. However, these approaches expect a standard, realistic human morphology, a luxury not afforded by the cartoonish, abstract forms found in NES games. Given this, the concept of a "run cycle" is beyond the capabilities of *Mappy*; however, there is other information encoded in the animation, for which it is essential to break down what exactly an animation is.

At the highest level, an animation is the communication of an activity. This activity is often representative of an ongoing state that an entity is in, e.g., running, jumping, standing still, etc. Some of these states are durative (e.g., running in *Super Mario Bros.* can continue for a long period of time) and their respective animations will loop many times. Some are "one-shot" animations, that play once (e.g., Kirby performs a flip at the apex of his jump in *Kirby's Adventure*) and these one-shot animations are often very simple (e.g, the jump animation in both *Super Mario Bros.* and *Mega Man* consist of a single static). These animations can be parameterized based on pertinent variables in the state that they are representing (e.g., the cycling of images in Mario's run in *Super Mario Bros.* speeds up as his x velocity increases). The key to extracting these animations, without reasoning about *what* they are actually conveying, is noticing the patterns, and how they repeat. If the communicative goal of an animation is to inform the player that an entity is in a specific state, when the player observes a pattern that

they have previously observed they infer that they are in the same – or at least similar – state. When the player observes a pattern that loops, they recognize it as one continuous state (e.g., the same 3 frames of animation looping in 2.13 reads as "running").

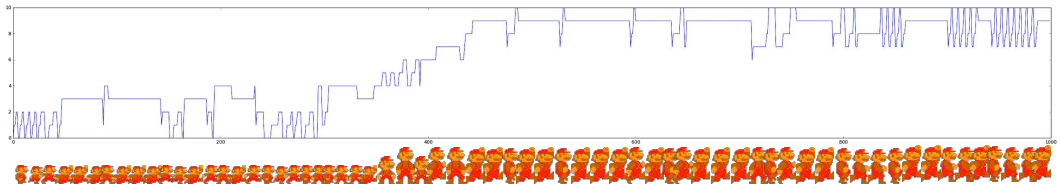


Figure 2.13: One thousand frames of Mario's animations in *Super Mario*. Each frame of animation is assigned a unique identifier, and the progression of these is shown in the top image. The current frame, sampled periodically, is shown on the bottom.

2.6 Mode Dynamics

There are modes of communication by which the states of entities are communicated to the player other than by "What animation is playing," namely the "physical" state of the entities, such as the position, velocity, and acceleration. Returning to the Mario run cycle shown in figure 2.8, when Mario is running, his position is updating in time. His speed is also updating in time, depending on whether he has reached his maximum velocity and whether the B button is pressed (which increases both the acceleration and the maximum x velocity). What would be detected as a single run state via the animation detection process in 2.5 is really four states – B button terminal velocity, no B button terminal velocity, B button acceleration, and no B button acceleration. By detecting the changes in acceleration and velocity, it should be possible to find these states.

These distinctions can be more subtle than can reasonably be surfaced by a player. For instance, one of the defining characteristics of *Super Mario Bros.* is its introduction of *jump control*, i.e.,the process of allowing the player to affect the duration of Mario's jump. This process is intuitive and player's discover it by holding the A button for different durations. However, while this process is intuitive to control, it is not instantly obvious as to what the actual underlying dynamics are. Does Mario's jump ascent stop the instant the button is released? Does releasing the button increase gravity? Does releasing the button add a sudden downward velocity to Mario? Does releasing button set Mario's velocity to a fixed value? All of these are possibilities, but a player is not assessing them as they are playing. They are instead developing an understanding of the "feel" [206].

Swink [206] takes a designer's look at what is meant when a person describes a game's controls as "floaty" or "sticky." Swink delves deep into a few example games to develop an intuition – albeit one guided by a very precise assessment of the mechanics – for the controls and physics in the game. Sudnow [192] takes a phenomenological approach to *Breakout* and *Missile Command* stating "as you watch the cursor move, your look appreciates the sight with thumbs in mind, and the joystick-button box feels like a genuine implement of action." [192] From this view point, the synaesthesia binding the control and visuals come together in one cohesive unit that we as humans understand. While a player may not have a perfect mental map of the internal states of the game, those states do exist, and seeing as they can be crucial for determining "game feel" it is important to extract them, as their communication of state is important.

2.7 Conclusion

As evidenced in the previous sections, there are a number of human processes that must be addressed for a computational system to make headway into the problem of annotating a game map via observation. *Mappy* is a system designed to address these issues, with a specific focus on action games for the Nintendo Entertainment System. The following chapter will discuss *Mappy* in detail, addressing each of these problems in turn.

Chapter 3

Mappy – A System for Map Extraction and Annotation via Observation

[1]

Mappy is a system designed to take in a game and a set of input traces for that game, producing an annotated map graph for that game. The game takes the form of a Read-Only Memory (ROM) file for the Nintendo Entertainment System (NES). The NES was chosen as the platform as it has a number of desirable properties – a large corpus of games, well understood hardware architecture, and a number of fast open-source emulators available. It is this last property that is particularly appealing, as an instrumented emulator allows for an important sidestepping of a fundamental problem, vision. Object detection approaches in computer vision systems have typically

[1] Portions of this chapter originally appeared in "Automatic Mapping of NES Games with Mappy" [138], "Charda: Causal hybrid Automata Recovery via Dynamic Analysis" [199], "Automated Game Design Learning" [141], and "What Does That ?-Block Do? Learning Latent Causal Affordances from Mario Play Traces" [193]

focused on the segmentation and detection of large, naturalistic images with orders of magnitude more pixel fidelity than that found in games. Where a few pixel margin of error in detection is acceptable when an object might be thousands of pixels, it becomes a more severe problem for an object made up of a few dozen pixels. Furthermore, these systems require supervised training, whereas *Mappy* runs in a unsupervised manner. By using an instrumented emulator, *Mappy* has perfect access to entity locations and object segmentation; however, as will be discussed in section 2.1 there are still a host of aspects of the annotation task that require careful consideration and technical handling.

The architecture for *Mappy* can be seen in figure 3.1. The input consists of:

- **Game** – a game ROM for the NES
- **Input Traces** – a set of input traces for the NES. An input trace is a list of frame-by-frame input commands for all controllers. For the NES, the possible inputs are the Start, Select, A, and B buttons and the direction(s) pressed on the directional pad – see figure 3.2. Note: an input trace can conceivably describe an input configuration not possible on the original controller hardware (e.g., the NES controller directional pad can only support a single directional input per frame as a product of the rigid plastic cross used, but it is possible to provide an input trace where any combination of directional inputs is provided in a given frame).

and the sub-processes of *Mappy* are:

- **Instrumented Emulator** – an emulator for the NES. *Mappy* currently uses FCEUX [47], but any emulator that provides access to the screen buffer, state

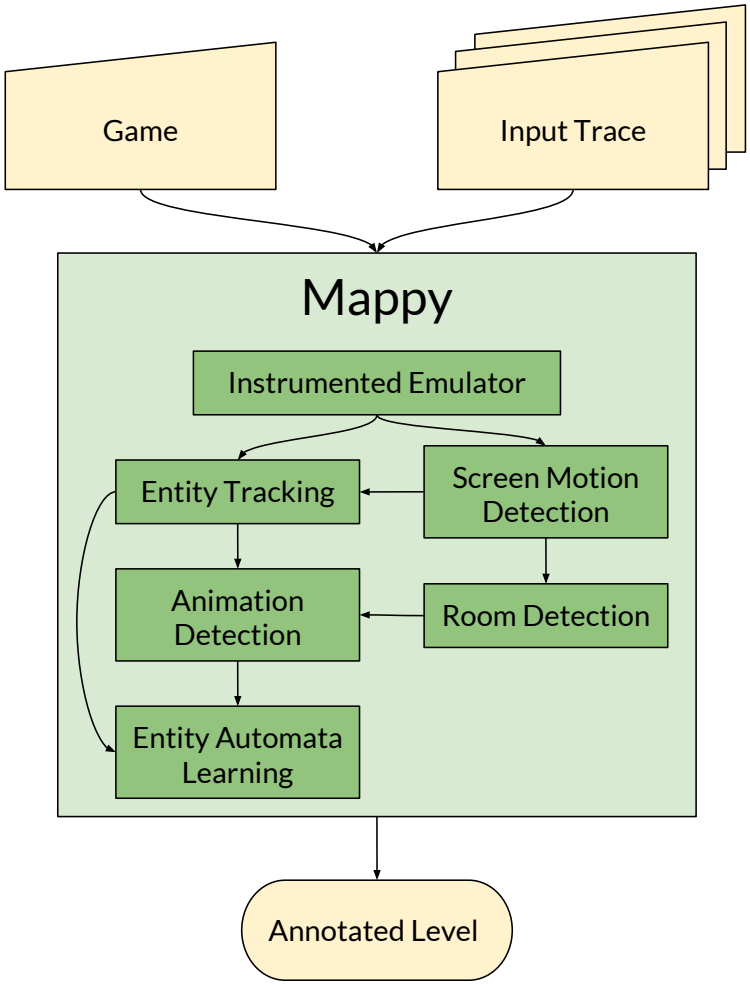


Figure 3.1: The architecture for *Mappy*. *Mappy* takes in a game and a set of input traces. These are passed to an instrumented emulator that provides entity locations which are used for determining overall camera behavior and entity tracks. The camera behavior is used to determine when a room begins or ends. The entity tracks are used to determine what animations occur at given points in time, and finally, along with said animation information, are used to learn an automata describing the properties and dynamics for each of the types of entities found. All of this information combined is used in the resultant map graph.

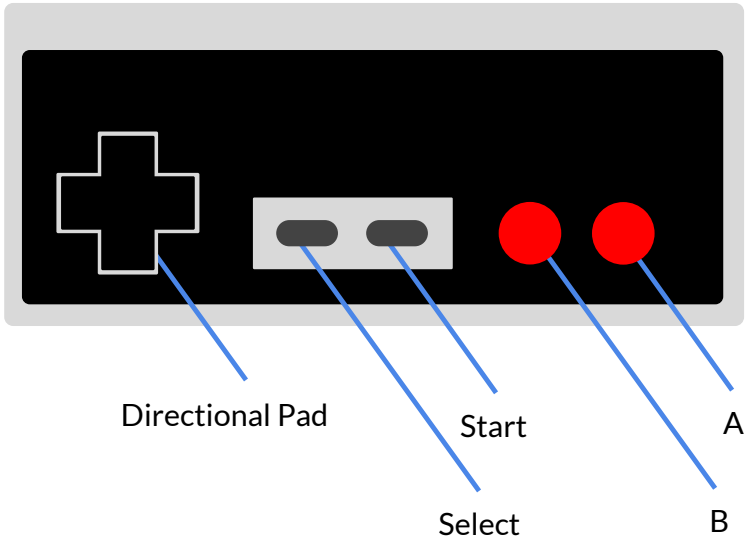


Figure 3.2: The controller for the Nintendo Entertainment System.

saving/reloading, and access to the Picture Processing Unit (PPU) (described in section 3.1) would suffice. The rest of *Mappy* is not specific to the NES and should generalize to two-dimensional games found on other systems – assuming engineering to extract out the relevant information.

- **Entity Tracking** – a suite of algorithms that take in the raw image locations per frame and provides a set of tracks for entities. This is described in more detail in section 3.2.

- **Camera Motion Detection** – An algorithm that takes in the screen buffer and PPU state and determines how the camera logic is being engaged at any point in time. This is described in more detail in section 3.3.

- **Room Detection** – An algorithm that take in the screen buffer, processed camera

motion, and information about player control and determines when the player has entered and exited a room. This is described in more detail in section 3.4.

- **Animation Detection** – An algorithm that take in the per frame image information on a per-track basis and determines the animation loops present for that entity. This is described in more detail in section 3.5.

- **Entity Automata Learning** – An algorithm that take in the per frame image information on a per-track basis and animations and determines what states are present for an entity, the dynamics of those states, and the transitions between states. This is described in more detail in section 3.6.

3.1 NES Architecture and Emulation

The NES is a nice platform as its hardware explicitly defines and supports the rendering of grid-aligned tiled maps (drawn at an offset by hardware scrolling features) and pixel-positioned sprites. The NES implements this with a separate graphics processor (the *Picture Processing Unit* or PPU) that has its own dedicated memory defining tilemaps, sprite positions (and other data), color palettes, and the 8×8 patterns which are eventually rasterized on-screen. During emulation, *Mappy* can directly read the PPU memory to access all these different types of data; we briefly describe the technical details below (referring the interested reader to [132]).

Although the PPU has the memory space to track 64 hardware sprites at once, there are two important limitations that games had to contend with: first, each sprite

is 8×8 pixels whereas game entities are often larger; and second, the PPU cannot draw more than eight hardware sprites on the same *scanline* (screen Y position). This means that sprites are generally used only on objects that *must* be positioned at arbitrary locations on the screen.

Static geometry, including background and foreground tiles, are not built of sprites but are instead defined in the *nametables*, four rectangular 32×30 grids of tile indices; these four nametables are themselves conceptually laid out in a square. Since the PPU only has enough RAM for two nametables, individual games define ways to mirror the two real nametables onto the four virtual nametables (some even provide extra RAM to populate all four nametables with distinct tiles). On each frame, one nametable is selected as a reference point; when a tile to be drawn is outside of this nametable (due to scrolling) the addressing wraps around to the appropriate adjacent nametable. Note that many rooms are much wider than 64 tiles—the room as a whole never exists in its player-visible form in memory, but is decompressed on the fly and loaded in slices into the off-screen parts of the nametables as the player moves around the stage.

Mappy remembers all the tiles that are drawn on the visible part of the screen, filling out a larger map with the observed tiles and updating that map as the tiles change. A *Mappy* room at this stage is a dictionary whose keys are a tuple of spatial coordinates (with the origin initially placed at the top-left of the first screen of the map) and the time points at which those coordinates were observed, and whose values are *tile keys*. A tile key combines the internal index used by the game to reference the

tile with the specific palette and other data necessary to render it properly (from the attribute table and other regions of NES memory). After *Mappy* has determined that the player has left the room (see Sec. 2.4), the map is offset so that the top-left corner of its bounding rectangle is the origin and all coordinates within the map are positive.

3.2 Operationalizing Entity Persistence

Given that the inductive bias of Operational Logics provide a good governing structure for abstraction, we see that Physics Logics cover the notion that presented game objects will be governed by a set of physical dynamics. This means that while the true information presented by the game, a series of flickering frames, has no notion of entity persistence, any useful abstraction must account for the fact that presented entities are not merely disjoint images in time, but rather represent the same entities across time with physical processes governing their motion. However, this presents the problem of "*How do we infer the persistence of animating entities across time, given disjoint frames of data?*"

Broadly, this is a target tracking [24] problem. Target tracking as a field has existed since the adoption of radio-based detection and tracking (i.e., RAdio Detection And Ranging [RADAR]). RADAR presents an operator with plots (representing high returns from a radio beam) that are disjoint in time (a problem similar to ours), and the early days of RADAR used humans to track aircraft. Kálmán [96] developed what is now commonly referred to as the Kalman filter to solve the problem of tracking an

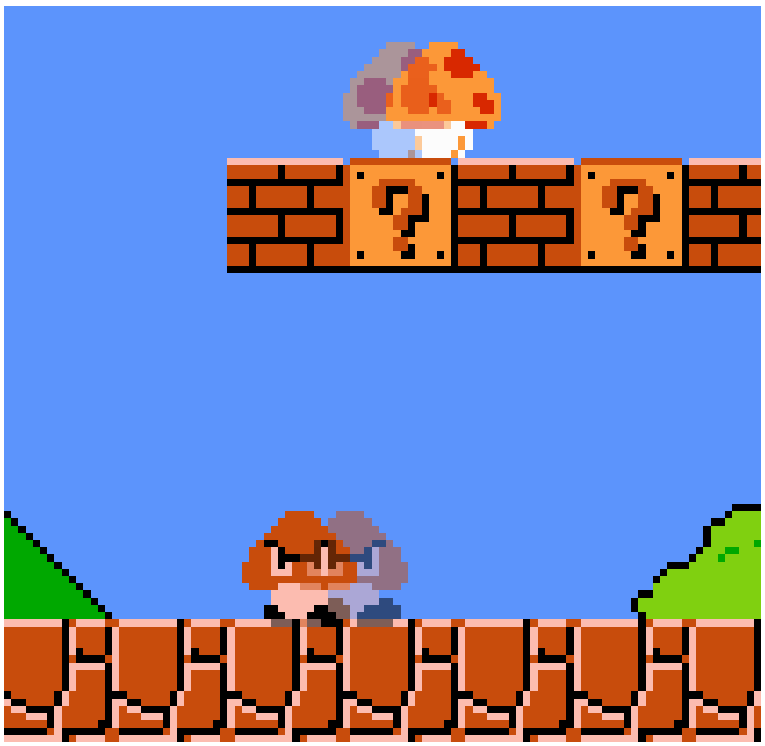


Figure 3.3: An exaggerated example of differences in two frames from *Super Mario Bros.*. The initial frame is slightly reduced in opacity for this illustration. This would lead to the apparent motion of the mushroom moving to the right and the lower Goomba moving to the left.

object given noisy measurements of what is believed to be a linear dynamical system. Extensions have been made for targets that might have multiple different modes of movement [172] and for tracking multiple targets [157].

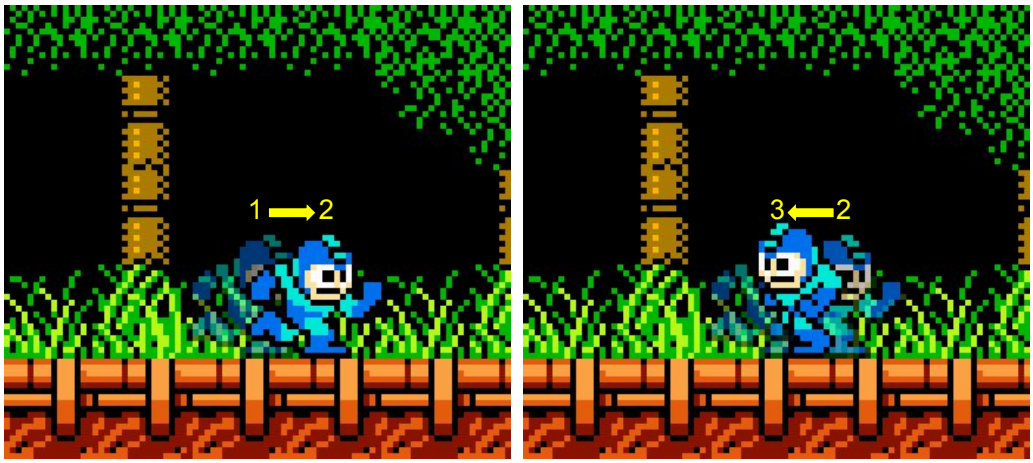


Figure 3.4: An exaggerated example of differences in three frames from *Mega Man*. The initial frame is slightly reduced in opacity for this illustration. In the left image, Mega Man starts at one position (frame 1) and moves right to (frame 2). On the right Mega Man abruptly changes direction and moves left from (frame 2) to (frame 3)

While the invocation of Physics Logics leads us to believe that the entities on screen follow a model of physical dynamics, the non-physical realm of video games can have drastically non-physical physics. While many game entities have momentum (e.g., all static entities, Mario in *Super Mario Bros.* etc.), there are many that do not. These are mostly due to abrupt state changes, be it player input (e.g., Mega Man), internal behavior logic (e.g., a boss in *Mega Man* "deciding" to start jumping or dashing), or game "physics" logic (e.g., spring board in *Super Mario Bros.*). Figure 3.4 demonstrates the instantaneous nature of Mega Man's ground motion. Even in games without instantaneous non-physical accelerations, their physics models might be

poorly captured by the homogenous physical equations assumed by a Kalman-family tracker. A model of aircraft dynamics will typically consider three modes of dynamics, straight-level flight, a fast high acceleration turn, and a slow, low acceleration turn. Mario's standing jump in *Super Mario Bros.* contains four distinct modes (stationary, button hold, button release, terminal velocity) and the full jump dynamics contains 22 different modes. In general, we do not want to assume any physical model, so a Kalman filter would be a poor fit. Furthermore, this tracking problem consists of hundreds of entities being tracked at any point in time, while standard multiple target tracking problems assume a relatively low density of targets (while air traffic control often needs to track hundreds of aircraft at any point in time, these are generally cooperative targets, broadcasting their own position and identity at each point in time, a luxury we do not share).

The instrumented NES emulator gives access to the nametable, which is an array of tiles commonly used as background, and the Object Attribute Memory (OAM), an array where each entry is the raw 3-tone image, the palette for coloring, whether the image is flipped horizontally, whether the image is flipped vertically, and the screen-space x and y coordinates. Order in the array need not hold any information about the identity of an entry (e.g., due to limitations in the number of sprites available per scan-line, some games will re-order the OAM such that any given sprite will not be invisible on 2 subsequent frames). These are updated on a per-frame basis. A given entity can have its image changed in an arbitrary manner per-frame (e.g., when Mario collects a star his colors constantly change, when Mario collects a mushroom he oscillates between

the normal and super Mario images for a brief period of time, and his images change as a normal part of animation frames).

The problem statement is thus:

There is a list of frames, F, where each $f \in F$ is a list of object tuples $\langle x, y, i \rangle$, where x, y are the screen-space positions and i is the image presented to the player. The goal is to construct a set of tracks, T, where each $t \in T$ is a list of tuples $\langle n, o \rangle$, where n is the index of the frame in F and o is the index to the object tuple in that frame f.

The underlying guiding principles governing the solution are:

- We want a minimal set of constraints on the underlying dynamics governing the "motion"
- We believe that it is more likely for an entity to move a small distance from its previous location, than a larger distance
- We want to make minimal assumptions about continuity of images, as images can change greatly in size or color

The solution in *Mappy* is as follows:

Initialize a list for active tracks, T_a, a list for inactive tracks, T. Iterate over each frame $f \in F$. For each frame, construct a bipartite graph, W, where each entity from the current frame, $e \in f$ is found on the left hand side, and each currently active track, $t_a \in T_a$ is found on the right hand side. The edges between all pairs of members of the left hand side and right hand side are formed with weight

$w = \frac{1}{\sigma\sqrt{2\pi}} e^{-(d)^2/2\sigma^2}$

where d is the distance between the entity on the left hand side and the last seen point in the track on the right hand side. σ is a hyper-parameter determining the scaling of this weight (higher σ being a flatter scaling, while a low σ being closer to a spike and slab weighting). This weight function is the probability distribution function of the Gaussian normal distribution of mean 0 and standard deviation σ. This has the interpretation that it is expected that each track will move in a Gaussian cloud around the previous position. Furthermore, for each point on the left hand side, a special *track start* node is constructed on the right hand side, with edge weight equal to w_s, a hyper-parameter determining the likelihood of starting a new track (a lower w_s means a lower likelihood of starting a new track, while a sufficiently high w_s will lead to a new track being formed each frame for each entity).

Given the bipartite graph, the goal is to assign each currently seen entity to a track, with the global maximum likelihood for the assignment, given the Gaussian prior on the entity movement. The maximum weight matching of Edmonds [57] is used, finding the optimal, maximal weight in $O(n^3)$ time.

After assigning each entity to a track (which could possibly be a new, previously empty track), a timeout parameter, o_t, is incremented for each track that did not receive an update. If a track's o_t is greater than o, a hyperparameter governing how long a track can coast, it is moved to from T_a to T. While it is relatively unlikely for a sprite to disappear, many games use a blinking effect where a sprite will disappear for a few frames (e.g., when Mega Man takes damage he blinks in a 4 frame cycle – 4 frames on, 4 frames off), so o can be relatively small in practice.

After all frames in F have been iterated over, any remaining tracks are moved from T_a to T, and a full accounting of entity persistence and motion across time is achieved. It should be noted that the motion at this point in time is only in screen-space, i.e.,what is presented via the screen, not world-space, i.e.,the ground truth as used by the simulation. To go from screen-space to world-space, we must disentangle the actual positions of entities from what is presented via the camera.

3.3 Inferring Camera Motion

Given that camera motion exists in games, the question becomes, "How can this understanding be operationalized?" As discussed, common-sense reasoning plays a huge part in how humans interpret the the motion of the camera, but that is beyond the scope of this dissertation. Instead, the motion of game entities will have to be analyzed to determine camera motion. There are a wide range of different behaviors for game cameras. As discussed by Itay Keren [101] there are a variety of factors that a camera must balance:

- What is the focus? – Is there a specific entity/set of entities that the camera is following? Is there a location that is pulling the camera's focus?
- How closely is the camera tracking? Is it locked to the focus entity's position? Is there a window that entity can move freely in before engaging the camera's motion?
- How quickly does the camera track? Is it always some fixed offset from the focus

entity or does it smoothly interpolate to that position?

- Are there different modalities for vertical and horizontal motion? Do these engage based on specific game world criterion?

and while these are all important questions when designing camera behavior, they are irrelevant for the more low-level question of "How do we detect camera motion?" The camera is moving, and this motion must be extracted, but the underlying logic of "Why is the camera moving?" or "How is the camera moving?" are not pertinent to the extraction of motion, while from a design stand point they might be relevant – i.e., if one were generating camera behaviors.

A common way of detecting camera motion in real-world scenes is via *optical flow* [19, 53]. Optical flow operates by finding the differences between two frames of video and tries to account for these differences given the construction of a flow field. This flow field represents how the pixels in the first frame would need to be translated to achieve what is seen in the subsequent frame. Optical flow works well in dense real-world scenes where there is a lot of texture provided to allow the algorithms to find and discern motion; however, the abstract, simplified scenes presented in NES games represent something of a unique challenge to standard optical flow techniques. Figure 3.5 shows an example of how a standard optical flow techniques fails in the face of *Super Mario Bros.* in discerning camera motion. The intuition in using optical flow to discern camera motion, is that the expectation is that most of the scene is stationary (i.e.,background) and that a small number of animate entities will be present on the

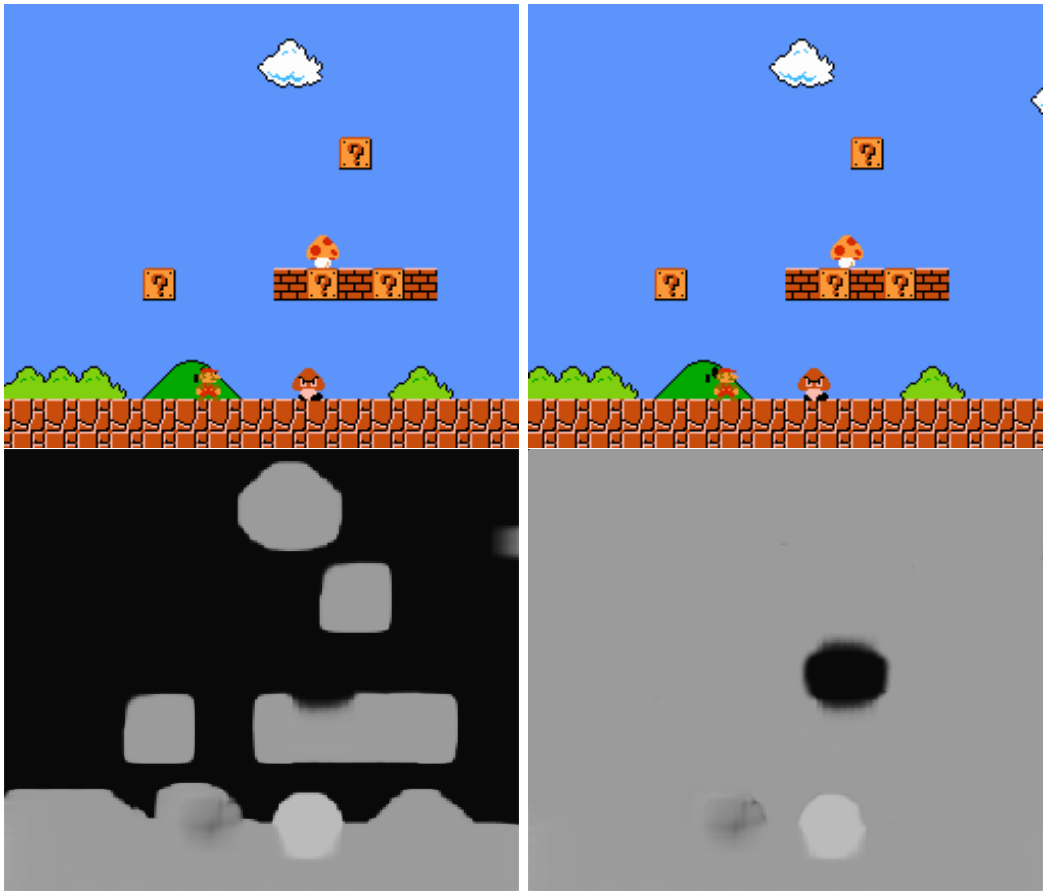


Figure 3.5: Two frames of *Super Mario Bros.*, the optical flow between them (as achieved via the technique described in "Two-Frame Motion Estimation Based on Polynomial Expansion" by Gunner Farneback in 2003" [91]), and the desired optical flow. Mario, the mushroom, and the camera move slightly to the right, while the Goomba moves to the left. Black in the optical flow images denotes no motion, while a higher intensity denotes more motion. The desired flow field should have all background elements with a slight (6 pixels) amount of motion, the Goomba with higher (9 pixels), Mario with slight (1 pixel), and the mushroom with no apparent motion; however, sky presents a large field in which no flow can be discerned.

screen. Therefore, the modal value of the optical flow should represent the opposite motion of the camera (e.g., if the most common motion in the scene is motion to the left, then the camera is panning right across the background). However, the sparse nature of NES game scenes mean that often the true nature of the motion is lost in a vast monotone swath. For instance, the solid color of the sky in *Super Mario Bros.* has no optical flow between frames, and since it makes up nearly three quarters of the screen real estate, an optical flow based technique will never discern any camera motion. However, while this simplified nature presents a drawback, a different simplified nature presents an opportunity.

Optical flow is useful for real world camera motion detection, due to the fact that a real world camera is traversing a three dimensional space, meaning the entire scene can change in scale and rotation. However, in a two dimensional game, the motion of the camera appears to be a sliding aperture over the scene. This means that per-pixel flow would, in the best case, determine the vector by which the first frame should be translated to achieve the second. This is in fact a problem solved via a different methodology in computer vision, *template matching*. Template matching is a method designed to solve the problem of: Given a source image, S, and a template image, T, find the position such that if you placed the template at that location within the source, it would maximize some measure of similarity. A number of similarity metrics are possible, including:

- Sum of Square Distance – $SSD(x,y) = -\sum_{x',y'}(S[x',y'] - T[x+x',y+y'])^2$

For all pixels find the square difference in intensity and sum

- Sum of Square Distance, Normalized – $SSD_n(x,y) = -\frac{\sum_{x',y'}(S[x',y']-T[x+x',y+y'])^2}{\sqrt{\sum_{x',y'} S[x',y']^2 \cdot \sum_{x',y'} T[x+x',y+y'])^2}}$

  Find the sum of squared distance between the two images, and then normalize by the root square intensities of the two images

- Cross Correlation – $CCorr(x,y) = \sum_{x',y'}(S[x',y'] \cdot T[x+x',y+y'])^2$

  For all pixels find the product of the two intensities and sum For all pixels,

- Cross Correlation, Normalized – $CCorr_n(x,y) = \frac{\sum_{x',y'}(S[x',y'] \cdot T[x+x',y+y'])^2}{\sqrt{\sum_{x',y'} S[x',y']^2 \cdot \sum_{x',y'} T[x+x',y+y'])^2}}$

  Find the cross correlation of the two images, and then normalize by the root square intensities of the two images

Considering the case of two identical images, all of these metrics will have a global peak (perhaps non-unique given the structure of the images) of goodness (a global maxima for the cross correlations, and a global minima for the square distances) when they are perfectly aligned, however, they will differ in their handling of non-identical images. Figure 3.6 shows an example template matching problem of finding a mushroom within a *Super Mario Bros.* scene, displaying the effects of the four different similarity metrics. Given template images that only differ in intensity, the sum of squares methods find matches at places most closely aligned to the average intensity (leading to poor results for darkened and lightened images). The standard cross correlation finds matches at places with high intensity (e.g., the clouds), but the normalized cross correlation metric has a match at the correct location no matter if the image is darkened or lightened. It is due to this behavior that the normalized cross correlation is the

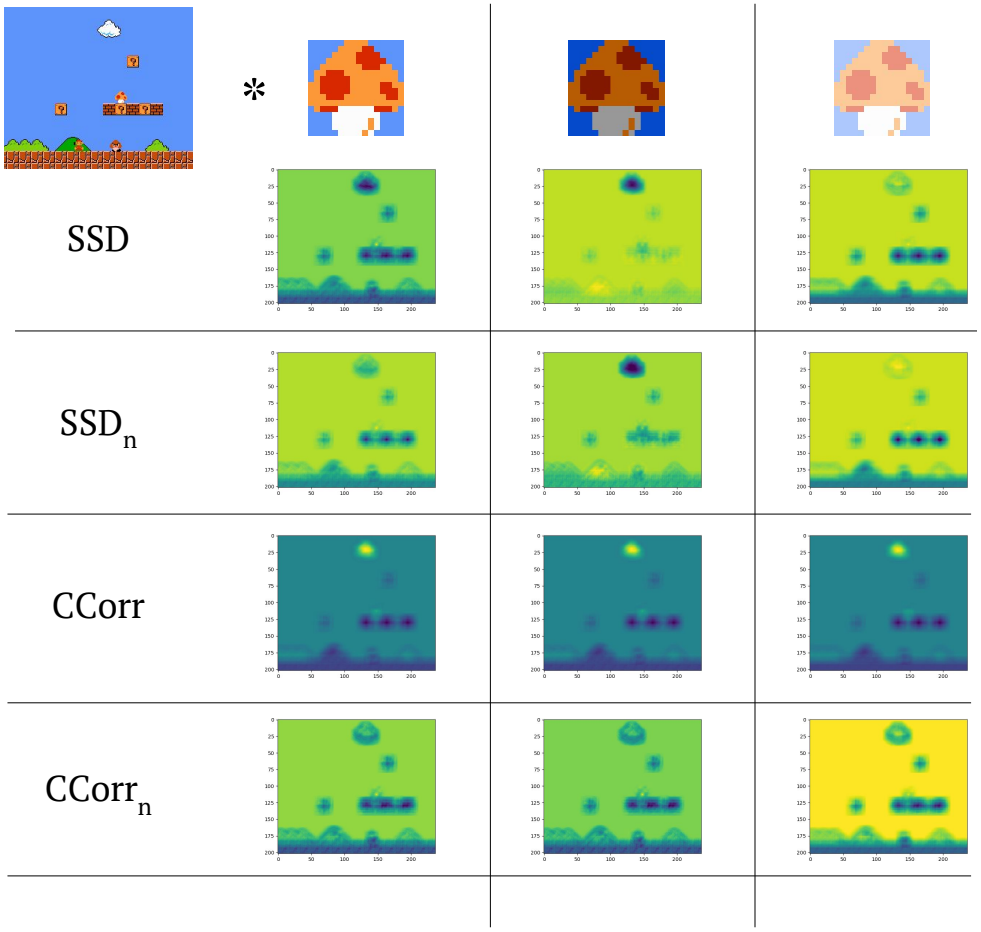
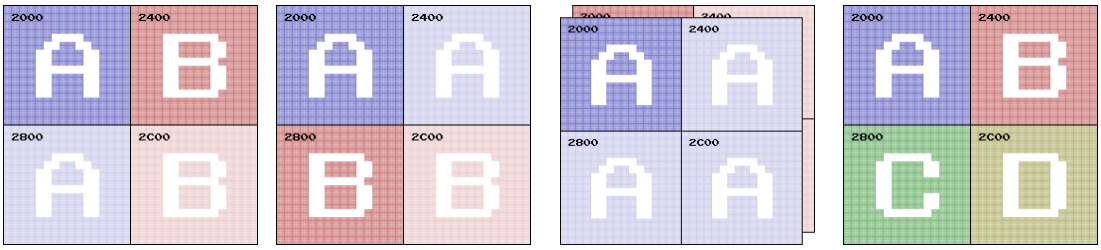


Figure 3.6: An example of template matching of two different template patterns (the normal, dark, and light mushrooms) on a single source image from *Super Mario Bros.* for the four different template matching metrics, $SSD, SSD_n, CCorr, CCorr_n$. Darkness indicates low similarity and lightness indicates high similarity. The sum of square distance metrics do well with the normal, light colored mushroom, but do poorly with the darkened mushroom (as its intensity is closer to that of the dark green bush near Mario) and the light (as its intensity is closer to the cloud). The non-normalized cross correlation method does poorly on all images, as the pure white cloud has a high cross-correlation with any image; however, the normalized version (which normalizes the expected return from the pure white cloud) finds identical peaks for both the normal, dark, and light mushrooms at the correct location.

similarity metric used going forward.

Template matching finds the optimal offset to place a template image in a larger source image but two frames of an NES game are identical in size. How then can template matching find the translation between two frames? It would be possible to embed the source image in some larger image with some sort of padding. This is relatively common in image based convolution techniques [13, 7], and is typically handled via zero padding (extending the image with pure black), reflection (mirroring the image on the borders), or periodic wrapping (e.g., moving past the border on the right wraps to the left, etc.); however, the nature of the nametables in the PPU gives a clean method for extension.



(a) Vertical mirroring (reproduced with permission from [12])

(b) Horizontal mirroring (reproduced with permission from [12])

(c) Single screen mirroring (reproduced with permission from [12])

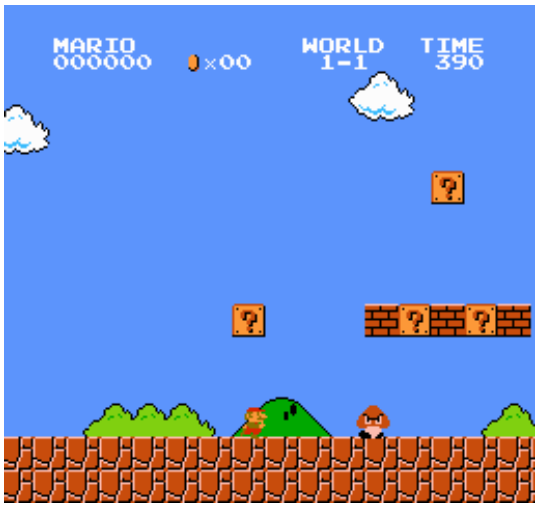
(d) Four screen mirroring (reproduced with permission from [12])

Figure 3.7: The four most common PPU mirroring modes.

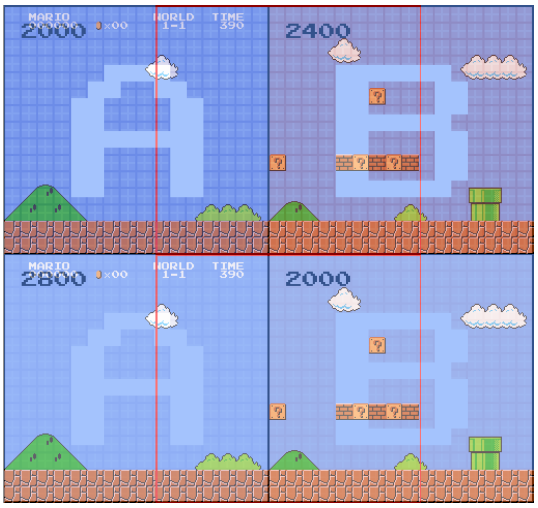
The nametables in the PPU hold the background tile information. They hold not just the information for what is presented on the screen, but actually contain information for a large surrounding area, depending on the mirroring mode. A number of different mirroring modes are possible, given the memory mapper present in the hardware, but the most common are horizontal mirroring, vertical mirroring, single screen

mirroring, and four screen mirroring. The PPU can map four screens at once: upper left, upper right, lower left, and lower right. Horizontal mirroring means that the upper left is identical to upper right and lower left is identical to lower right. Vertical mirroring means that the upper left is identical to lower left and upper right is identical to lower right. Single screen mirroring is fully mirrored with all four quadrants being identical. Four screen mirroring is the absence of any mirroring with each quadrant being unique. These mirroring modes can be seen in figure 3.7. Horizontal mirroring is found, perhaps unintuitively, in games featuring vertical scrolling, while vertical mirroring is found, similarly unintuitively, in games with horizontal scrolling. While unintuitive, this makes a great deal of sense when one considers that two screens arranged left-to-right (i.e.,vertical mirroring) can be smoothly scrolled across from left-to-right. The nametable provides a larger image, extended from the original, with no need to guess at the border behavior (which would almost certainly be lossy due to the fact that zeros, reflecting, and wrapping are unlikely to capture the true border behavior), within which the template, in this case the screen as it is presented to the player, can be found. An example from *Super Mario Bros.* can be seen in figure 3.8.

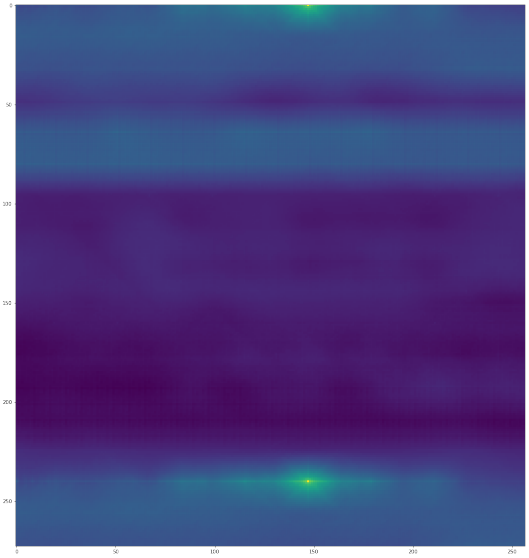
Given the location of the screen within the nametable, it is trivial to record the difference between translations on a frame-to-frame basis. E.g., if the optimal offset is found to be $(148, 0)$ pixels on frame 1 and $(150, 0)$ pixels on frame 2, we can conclude that the camera moved two pixels to the right. With these frame-to-frame camera movements, it is then trivial to deconvolve world-space coordinates, p_w, for entities via the camera positions, p_c, and the screen-space coordinates, p_s, as



(a) The screen as presented to the player.



(b) The nametable (with mirroring mode overlay) as found in the PPU's memory. The red box indicates the true position of the screen as embedded within the nametable.



(c) The result of template matching the screen on the nametable.

Figure 3.8: The process of template matching the screen (a) to the source image of the nametable (b) and finding the optimal matching (c). Note that there are two optimal translations, one for upper and lower mirrorings. To account for this, only the first translation (top-to-bottom, left-to-right) is kept).

$$p_w = p_c + p_s$$

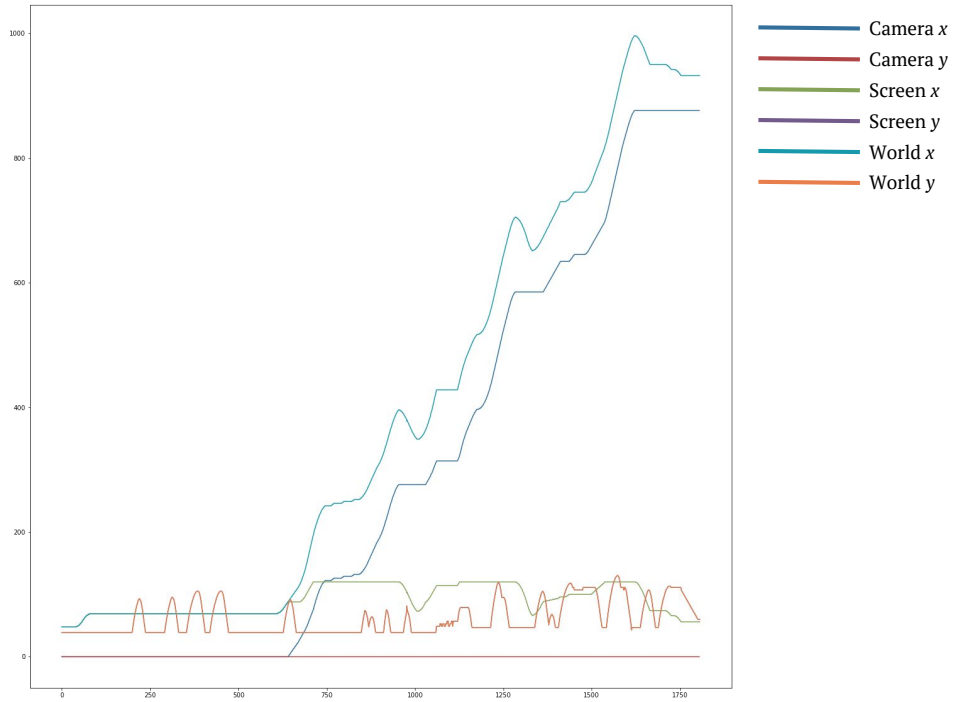


Figure 3.9: A demonstration of incorporating camera motion and screen space position to acquire world space position. Note that the camera has no motion in the vertical axis, so there is no difference between the screen and world y's (hence the occlusion of the screen y).

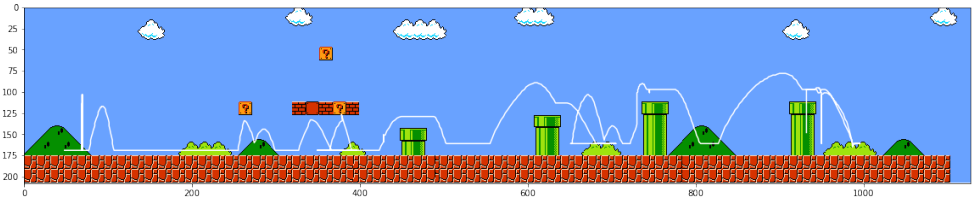


Figure 3.10: The result of extracting Mario's path via the camera motion.

Figure 3.9 shows the results of applying camera positions to the screen space coordinates for an example track of Mario in *Super Mario Bros.*, for which the resultant track overlaid on the room can be seen in figure 3.10.

Given the ability to find the world-space coordinates for all entities, it is now possible to discern the global structure of the simulated world to extract the rooms from the game.

3.4 Determining Room Transitions

The algorithm for determining when the player starts transitioning, when the player is transitioning, and when the player has completed transitioning is as follows. Initialize the time since player last had control, t_{pc}, the accumulated motion since the player last had control, $x_a \& y_a$, to 0.

If the player does not have control, then increment t_{pc} and add the current detected camera motion x_c, y_c to x_a, y_a; however, if the player has regained control, set t_{pc}, x_a, y_a to 0.

If $t_{pc} > t_g$ and either $x_a > x_s$, $y_a > y_s$ then set a flag indicating that the player is in an interstitial state and a flag indicating the possibility of an interstitial state is unset. t_g is a parameter acting as a threshold on determining whether the player is possibly in an interstitial state, x_s and y_s are thresholds determining how much motion must occur for it to be believed that the player has entered a traversal interstitial. The checks on the x and y accumulators are meant to determine if the screen has seen a large amount of motion (indicative of scrolling into a new room) as opposed to no motion (as occurs during the loss of control when the player collects a power-up in *Metroid*) or a stochastic "screen-shake" (as occurs in *The Young Indiana Jones Chronicles*). If

$t_{pc} > t_g$ but all of the other checks fail, then a flag indicating potential interstitial is set. If the potential interstitial flag is set and $t_{pc} < t_g$ then the flag is unset.

Algorithm 1 The algorithm for determining room transitions.

```
1:  procedure ROOM TRANSITIONS
2:      t_pc ← 0
3:      x_a ← 0
4:      I ← false
5:      for Frame do
6:          if Player Has Control then
7:              t_pc ← 0
8:              x_a ← 0
9:              y_a ← 0
10:         else
11:             t_pc ← t_pc + δt
12:             x_a ← x_a + x_c
13:             y_a ← y_a + y_c
14:         end if
15:         if t_pc > t_g ∧ (x_a > x_s ∨ y_a > y_s) then
16:             I ← true
17:         end if
18:         if Player Has Control ∧(I ∨ d > d_t ∨ ccorr_{f−1→f} < ccorr_t) then
19:             I ← false
20:             Record the previously seen room
21:         end if
22:     end for
23: end procedure
```

If the player has regained control and the interstitial flag was set, or $d > d_t$, or the cross correlation between pixels in the current screen and the previous seen is less than $ccorr_t$ then it is believed that a new room has been entered. d is the number of background tiles that have changed between the previous and current frames and d_t is a threshold for the difference. All recorded frames are stepped through in this manner, leading to a segmentation of which room the player is in at each point in time in the play trace.

For some games, there are continuous transitions, which *Mappy* would make treat as one room – it remains future work to see how this would affect generation.

3.4.1 Automatic Mapping

Given the flow from raw frame-by-frame positions → screen-space entity tracks throughout time → world-space entity tracks throughout time → entity tracks segmented by which room they fall in, it is now possible to extract room maps from games. Given the nature of maps on the NES, a grid of tiles is the most reasonable format as the bulk of the entities are the tiles found in the nametable. To construct the map, a dictionary, $\mathcal{M}_r$, is formed for each room, r. On each frame, all of the tracks have their current positions, x, y, embedded within the grid (8×8 pixel fidelity for the NES) and the image displayed, I, is appended to an array (initialized for each grid location in the dictionary if this is the first time that location has been encountered), i.e., $\mathcal{M}_r[\lfloor x/8 \rfloor, \lfloor y/8 \rfloor]^\frown \langle I \rangle$. After this process is completed, a map is constructed with each element of the map being a sequence of images seen over the duration of that entity's visibility, i.e., its *animation*. This leads to the next distinction, "What is an animation, and how do people recognize animation?"

3.5 The Detection of Animation Cycles

As discussed in section 2.5, when the player observes a pattern that loops, they recognize it as one continuous state (e.g., the same 3 frames of animation looping in 2.13 reads as "running"). The goal, therefore, is to find the repetitions and loops that

best cover the observed image frames.

Globally, this is an compression problem, with the goal being the compression of a sequence of images into a sequence of animations, with the goal being the minimization of the space required to describe the sequence and the space required to describe the unique animations. Given a sequence of length n with the set of unique frames, F, segmenting it into 1 animation of length n, will take up $n \log |F| + 1$ bits of information (1 for the id of the 1 animation, $n \log |F|$ bits [there are n frames in the animation, each of which requires $log|F|$ bits to encode] to describe that one animation). Conversely, segmenting it into n animations of length 1 will take $n \log n + n \log |F|$ bits (there are $|A|$ different animations, so the sequence of animations takes $n \log n$ bits and each of the n animations takes $\log |F|$ bits to encode). These degenerate cases are far from what is to be expected, as these animations are intended to convey information, so they are unlikely to be purely, stochastically information dense. Figure 3.11 shows the ground truth for the animation states over the 1000 frames of play in *Super Mario Bros.*. There are eight different animations used over this duration, and the ideal goal would be to segment this sequence to extract these eight animations.

Given the global nature of this problem, all tracks that are thought to reasonably be the same type of entity are used to find the animations. This is achieved by finding the categorical distributions of the displayed images for each track. Those tracks that are found to be similar are merged. The similarity metric used is the Bhattacharyya distance,

$$D_B(p, q) = -\ln\left(\sum_{x \in X} \sqrt{p(x)q(x)}\right)$$

a distance metric for discrete probability distributions (0 when two distributions are identical, ∞ when two distributions share no support). In practice, a D_B threshold of 0.1 was found to properly group together entities, correctly avoiding grouping tracks those that shared support but were dissimilar (e.g., in *Super Mario Bros.* the animation of a brick that is hit involves turning into a sky tile, but it is unreasonable to say that a brick is the same as the sky even though they share the same image for a duration of time).

Furthermore, to reduce the size of the problem space, and to smooth over irregularities in displayed animations, a preprocessing Run Length Encoding (RLE) step is performed. The RLE takes a series of data and encodes it as pairs of identifiers and the duration associated with that identifier. E.g., the sequence $(A, A, A, B, B, B, C, C, C)$ would be encoded as $(< A, 3 >, < B, 3 >, < C, 3 >)$. As mentioned, part of the goal of this is to reduce the noise found from irregularities in the animations. E.g., in *Super Mario Bros.* the animations of the ?-blocks is normally (8-frames of image 1, 8-frames of image 2, 16-frames of image 3), but, occasionally, this slips to (8-frames of image 1, 7-frames of image 2, 17-frames of image 3). Reasonably, one would assume that this is a slight irregularity in the clock-timing of the CPU and not indicative of some important change in animation, especially since it will quickly revert to the standard (8,8,16) cycle after the faulty cycle.

The problem definition is:

Given a sequence of unique identifiers, S, find a set of animations, $\mathcal{A}$, where each animation is a sequence of unique identifiers and a sequence of these animations,

A, such that each element of S is covered by an animation, minimizing

$$O = |\text{RLE}(A)| + \sum_{a \in \mathcal{A}} |a|$$

At first blush, this would appear similar to classic segmentation problems that seek to minimize some cost related to the number of segments, $|\text{RLE}(A)|$, and the cost of those segments, $\sum_{a \in \mathcal{A}} |a|$; however, the presence of $\sum_{a \in \mathcal{A}} |a|$ precludes such techniques, as the global cost for the animation can not be considered for each segment separately (consider two identical modes of lengths n_1, n_2 respectively – treating them separately would have a cost proportional to $\log n_1 + \log n_2$ which is greater than treating them as one mode $\log n_1 + n_2$). Exhaustively, this problem would need to consider all 2^n possible segmentations. However, in practice, genetic search is capable of finding optimal, or near optimal, segmentations in reasonable time frames.

- **Genotype** – A sequence of integers, s.t. the sum of the sequence is equal to the length of S, i.e.,the grouping together of segments. E.g., given $S = (0, 1, 2, 0, 1, 2, 3, 3, 3)$ a few valid genotypes would be $(3, 3, 3), (9), (2, 1, 2, 2, 2)$

- **Mutation** – Segments can either be split, producing two segments such that their sum is equal to the original segment, or merged (two segments are replaced with a single segment such that the sum of the original segments is equal to the new segment.

  E.g.,

  - *split* – $(3, 3, 3) \rightarrow (1, 2, 3, 3)$
  - *merge* – $(3, 3, 3) \rightarrow (6, 3)$

- **Crossover** – Given two segmentations, A, B, randomly choose a cross over sum, s, in between 1 and n. Find the segments in A, B corresponding to that index, a, b. If that s is not equal to the sum of all segments up to and including a, then split a to a_l, a_r s.t. $a = a_l + a_r$ and the sum of all segments up to including $a_l = s$ – the index of which is i_a. Do the same for b, which will have a corresponding index i_b. The two children are then $AB = A[0...i_a] + a_l + b_r + B[i_b + 1...k]$ and $BA = B[0...i_b] + b_l + a_r + A[i_a + 1...k]$.

  E.g., If $A = (1, 1, 1, 2, 1, 3), B = (3, 3, 3)$ and i is chosen to be 5 then $AB = [1, 1, 1, 2, 1, 3]$ (with $(1, 1, 1, 2)$ coming from A, 1 being the split portion of a 3 from B and 3 coming from B) and $BA = [3, 2, 1, 3]$ (with the first 3 coming from B, 2 being split from a 3 in B, and $(1, 3)$ coming from A), with $a_l = \{\}, a_r = \{\}, b_l = \{2\}, b_r = \{1\}$

Unlike a standard genetic search, the structure of the genome provides a heuristic for the search – given the desire for the segmentation to use repeated animations, rare animations are more likely to be the incorrectly segmented animations. To wit while it is certainly possible that an animation might only play once, if an animation shows up only one time, it is more likely that it should be split or merged with one of its neighbors than an animation that is seen many times. This allows the biasing of the mutation operation in the choosing of which id to mutate. For each index, i, count the number of times its animation is seen, c_i. Select an index with probability:

$$P(i) = \frac{e^{\lambda c_i}}{\sum e^{\lambda c_j}}$$

Where λ is a negative real number, that determines how close to uniform this distribution is – the closer λ is to 0, the more uniform, and closer λ is to $-\inf$ the closer it is to deterministically choosing the index with the least seen animation.

The genetic search uses tournament selection whereby the population is ranked according to their score. For each selection, the ranked members are traversed in ascending order where each is selected with probability p_s. i.e., The most fit member of the population will be selected with probability p_s, the second most with probability $p_s(1-p_s)$, the third with probability $p_s(1-p_s)^2$, etc. This process is repeated k times.

After selection, the remaining $n-k$ members of the subsequent population are generated via the mutation and crossover processes, each with their respective rates p_m, p_c. This process is repeated for K generations, with the most fit member of the final population being the resultant answer.

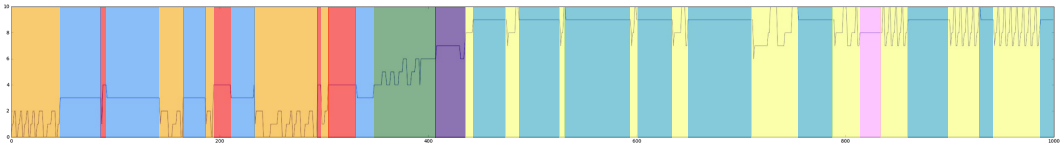


Figure 3.11: One thousand frames of Mario's animations in *Super Mario*. Each frame of animation is assigned a unique identifier, and each region of color is a different animation

For the 1000 frames of Mario animation shown in figure 2.13 the genetic search found the optimal segmentation in 70.4 generations on average, over the course of 10 trials, with $n = 1000, k = 150, p_s = 0.15, p_m = 0.25, p_c = 0.15$. The resultant segmentation can be seen in figure 3.11. With *Mappy* capable of determining what animations are occurring (and reoccurring), it is capable of detecting *when* an animation occurs. This is a critical step for the final, most important fact to be gathered, the causal reason

for *why* a state change occurs.

3.6 The Extraction of Dynamics Modes

Before discussing the extraction process, it is important to find a structure that is capable of capturing the complex dynamics in a general enough manner. A common structure found in games are Finite State Machines (FSMs). FSMs are defined as a set of states, the edges between the states, and the conditions required to traverse those edges. FSMs are an intuitive representation, popping up in many different game design tools [89, 1]. While these systems are often thought of and referred to as FSMs [8], they are not classical FSMs as their states have behaviors and themselves hold state (e.g., time in state, position, etc.). In general, these systems are Turing complete, which, while useful for expressiveness, is problematic for other concerns such as decidability.

Hybrid Automata (HA) are a class of automata that represent a natural extension to classical FSMs, namely that the states have a set of continuous variables that update via per-state systems of Ordinary Differential Equations (ODEs). While, obviously, not as expressive as Turing machines, the vast majority of game entity dynamics can be represented as Hybrid Automata, without loss of generality (and those that cannot can often be approximated with minimal loss of accuracy). HAs have a number of desirable properties, not the least of which is their intuitive nature and the relative ease of interpretation. Beyond interpretability, despite the general undecidability of many HA properties, it is possible to constrain models or carefully choose semantics

to obtain different analysis characteristics: discretizing time or variable values evades undecidability by approximating the true dynamics [95]; keeping these continuous but constraining the allowed flow and guard conditions admits geometric analysis [62]; and one can always merge states together to yield an over-approximation, producing smaller and simpler models. There are also composable variations of hybrid automata that admit compositional analysis [20] as well as a logical axiomatization [148], not to mention the body of tools and research that already exist for synthesizing control policies, ensuring safety, characterizing reachable areas, et cetera.

Extracting automata from observation is a common goal for which there exist a number of approaches. Ersen and Sariel-Talay [59] extract FSMs via observation of the interactions of entities in *The Incredible Machine*; however, their approach does not learn the dynamics that occur within different states. HyBUTLA [135] aims to learn a complete automaton from observational data. HyBUTLA seems able to learn only acyclic hybrid automata, since it works by constructing a prefix acceptor tree of the modes for each observation episode and then merges compatible modes from the bottom up. Moreover, HyBUTLA assumes that the segmentation is given in advance and that all transitions happen due to individual discrete events, presumably from a relatively small set. Santana et al. [159] learned Probabilistic Hybrid Automata (PHA) from observation using Expectation-Maximization. At each stage of the EM algorithm a Support Vector Machine was trained to predict the probability of transitioning to a new mode. However, their approach required a priori knowledge about the number of modes, a strong requirement in the very open ended realm of arbitrary entity behavior.

The closest work to that presented below is that of Ly and Lipson[115] which used Evolutionary Computation to perform clustered symbolic regression to find common modes with the Akaike Information Criterion uses to penalize model complexity. However, unlike *Mappy* their work assumes a priori knowledge about the number of modes. Moreover, since their work assigns individual datapoints, not intervals, to a mode, their approach can only model stationary processes.

The "game engine" learning of Guzdial et al.[72] sought to learn a set of rules – i.e.,an "engine" – to predict the next frame of a video for a game. In many ways, the underlying goal is the same, but the approach is completely different. Their approach seeks to learn all rules at once, which results in the need for a lot of rule pruning. The orthogonality brought about by considering entities persistence across time, animations, physical dynamics, and the rules governing their change as separate is instead put into one giant soup that the system must disentangle, learning (and later revising) rules such as "an initial rule that moves Mario right might include the condition that he has a cloud above him, a bush to his left, etc."[72] Furthermore, their approach only handles videos of games, and as such the "control rules" learned are instead probabilistic rules of the form that "leads to a branch of two possible frames for each control rule (the player chooses ... or doesn't),"[72], whereas *Mappy*, due to its direct connection with actual live play, learns the exact input that leads to changes in dynamics.

Given that HAs are an attractive target format for representing entity dynamics and states, *Mappy* must

- Detect when a given state is active
- Detect the dynamics of a given state
- Detect the causes for entrance to or exit from a state

subject to

- No prior knowledge about the number of possible states
- No prior knowledge about the dynamical equations of a state
- No prior knowledge about when a state begins or ends

with the goal being the optimal extraction of the true Hybrid Automaton governing the dynamics of the entity. As interpretability is a key consideration in the choice of Hybrid Automata, the notion of optimality has to consider some notion interpretation. For instance, accuracy as an optimization criterion is a poor fit, as the most accurate model would encode every data point as its own state with constant behavior equivalent to that point, which is highly unlikely to be a representative model for that entity. Instead, *Mappy* turns to two different model selection techniques to find an optimal configuration of states, the Bayesian Information Criterion (BIC) [164] and the Minimum Description Length (MDL) [190]. BIC represents the problem of determining which model should be selected from a Bayesian viewpoint. Given a fixed penalty for selecting the wrong model, as the number of data points, n, approaches infinity, the optimal model will be the one that minimizes the penalty of:

$$-\log(\mathcal{L}(m|d)) + \frac{\dim(m)\log(n)}{2}$$

where $\mathcal{L}(m|d)$ is the likelihood of the the model, m, given the observed data d. Asymptotically, if the true model is under consideration in the set of models, then BIC will select the correct model with probability approaching 1 [223].

MDL is a similar measure in form, although it comes from an information theoretic grounding. The governing principle of MDL is to find a way to encode the information found in the original dataset such that the encoding requires the minimal amount of information. This is broadly done by finding a model, m, such that the amount of information required to encode the model in addition to the amount of information required to encode the differences between model predictions and actual data are minimized. This results in a penalty of:

$$-\log(\mathcal{L}(m|d)) + \dim(m)(1 + \tfrac{\log(n)}{2})$$

i.e.,the amount of information required to encode the differences (the log-likelihood of the model given the data) and the amount of information required to encode the model. The difference between MDL and BIC comes from their handling of model complexity. BIC scales the dimensionality of the model by $\frac{\log(n)}{2}$ as the asymptotic penalty for a more complex model, whereas MDL encodes the model via the following scheme. For each parameter, θ_i, define a base parameter, θ_0 (commonly 0). If $\theta_i = \theta_0$, use 1 bit to encode this. If not, use the location in a grid spaced at $\sqrt{n}$ to encode the position of θ_i, requiring $\frac{\log(n)}{2}$ bits of information.

A quick digression is necessary to discuss what exactly is meant by the number of parameters found in a model. Generally, this would be the number of trainable variables to be found in a model, i.e.,the number of weights in a neural network or linear

regression. However, in the combined context of a linear regression and the presence of models with zero trainable variables, this interpretation falls apart. Consider the case where there are zero trainable parameters, most commonly due to the presence of a null model that predicts a constant (most likely 0) value. In this case, due to the fact that there are zero parameters, both BIC and MDL are reduce to just the negative log-likelihood term, $-\log(\mathcal{L}(m|d))$. Thus, there is no penalty for the addition of a segment, so long as the likelihoods of the two segments is the same as if there was one segment, which violates the assumption that parsimony should be valued. However, from an information theoretic viewpoint, it is not the case that the null model has no parameters, because at some level, it must be encoded that that model was the one selected for a given segment. Thus, if there are k different models to be selected between, it takes $\log_2(k)$ bits to encode this model selection. Consider the case when there are two potential models, $y = 0, y = 1$. In this case it would take 1 bit to determine which model is chosen for a given segment, so each model should be considered to have 1 parameter.

Broadly, the goal of *Mappy* is to find the HA, m, that best meets one of these criteria, given an input trace of entity positions, such that the model has the optimal balance between description of the true dynamics ($-\log(\mathcal{L}(m|d))$) while maintaining interpretability ($\lambda\dim(m)$). The first step in this process will be determining which data points of the trace should be grouped together as being generated by one state and determining the system of equations governing the dynamics of that state.

While the goal is to have the minimal amount of fore-knowledge about the

types of models that can be found, in practice it is infeasible to search from the set of all possible mathematical expressions. Given this, *Mappy* considers a set of model templates, $\mathcal{M}$, from which the dynamic models can be pulled from. For instance, in the case of learning the jump dynamics of Mario from *Super Mario Bros.* the following model templates are considered:

- $\dot{y} = 0$ – the null model where in the y-velocity is 0
- $\dot{y} = c$ – the model of constant $y - velocity$
- $\dot{y} = a \cdot t$ – the model of constant $y - acceleration$ with an initial y-velocity of 0
- $\dot{y} = a \cdot t + c$ – the model of constant $y - acceleration$ with an initial constant y-velocity

Given a data trace, $d = \{p_0, p_1, ..., p_n\}$, the goal is to find the switchpoints, $S \subset \mathbb{N}, s \leq n, \forall s \in S$, that signify when a new mode has been entered, such that one of the penalty criteria described above is minimized. To reduce the size of the search, only a specific subset of the full data trace are allowed to be considered as potential switchpoints. Operational Logics provide a guideline for reasons why a change in state might occur, and these allow the reduction in the search space. The simplest are those related to player control. Given the previous discussion of player control in section 2.4, it is obvious that changes in a player's input can result in the state communicated to players. As such, it is stands to reason that the times associated with the change in the player's input should be important times, as these are likely to be causally associated

with changes in dynamics. Now, there are certainly plenty of inputs that might not result in a change in state (e.g., if Mario is in the middle of a jump in *Super Mario Bros.* then pressing up or down will have no effect on the jump), so it is important to note that these are only *potential* switchpoints. More significantly, there are heuristics for when a change in dynamics are likely to have occurred:

- **Constant Endpoints** – Periods of constant velocity that then abruptly change are very likely to be caused by a major shift in dynamics
- **Zero Crossings** – It is common that a change in velocity that crosses zero, i.e., it goes from positive to negative or vice-versa – is the result of a change in the dynamics (e.g., Link walking left and then right) or might be associated with the change in the dynamics (e.g., the gravity value for Mario changes when his y-velocity goes from upward motion to downward).

While the number of switchpoints considered with these reductions is still $\mathcal{O}(n)$, in practice this results in a reduction of the search space by a large constant factor (e.g., in the consideration of 1000 frames of *Super Mario Bros.* this results in the reduction of potential switch points from 1000 to 203 – resulting in a 2426% speedup).

Let T be a table of model parameters with one entry for each interval i, j and model template m. Then we define T's entries as:

$$T[i, j, m] = \text{train}(m, d[i : j]) \quad \text{s.t.}$$

$$1 < i < j < n, m \in \mathcal{M}$$

where $\text{train}(m, d[i:j])$ is the method for training the model template m on the data found in the segment $d[i:j]$, n is the number of potential switchpoints, $\mathcal{M}$ is the set of model templates, d is the dataset, and $T[i, j, m]$ is the model of template m trained on data from the interval of i to j. i.e., fill the table T with all possible sub-models for all possible sub-intervals of the data. For this work, all model templates are multivariate regressions, but the approach is general enough to work with any approach that supports a likelihood function $\mathcal{L}(m|d)$.

The cost for a given model m for sub-interval i to j is therefore:

$$C[i, j, m] = -\log(\mathcal{L}(T[i, j, m]|d[i:j])) + \text{pen}(m, d[i:j])$$

given the penalty criterion pen. Given the aforementioned criteria, the possible penalty functions are

$$\text{pen}_{\text{BIC}} = \dim(m)\log(n)/2$$

and

$$\text{pen}_{\text{MDL}} = \dim(m)(1 + \log(n)/2)$$

For all segments that end at point j, find the optimal model and segmentation that leads to that point. $O[j]$ is the optimal cumulative cost of models across segments

up to datapoint j.

$$O[j] = \begin{cases} 0 & \text{if } j = 0 \\ \text{argmin}_{0 \leq i \leq j, m} \; C(i, j, m) + O[i-1] & \text{if } j > 0 \end{cases}$$

Following the classic dynamic programming algorithm for segmented regression [27], the optimal segmentation, S, is then found by working backwards through O. Starting with $j = n$, find the optimal segmentation ending at j, find the starting index of that segmentation, i, and repeat this process until $i = 0$. Thus, the optimal segmentation is found.

However, a key issue at this point is that while this is the optimal segmentation, it is still not the optimal description of the states. E.g., consider the toy data trace of $[0, 0, 0, 1, 1, 1, 0, 0, 0]$. An optimal segmentation would be $i = 0, j = 2 : y = 0, i = 3, j = 5 : y = 1, i = 6, j = 8 : y = 0$. However, the optimal configuration would consider this an automaton with 3 parameters, one with two states that set $y = 0$ and one state that sets $y = 1$. This feels unlikely, as the intuition is that the first and last states are one and the same. This means the global structure must be considered that best describes the data. This is very close to the structure of the problem described in section 3.5, so again genetic search is used to find the optimal configuration, however with the optimization score being $-\log(\mathcal{L}(A|d)) + \text{pen}(A, d)$, where A is the automaton.

While this search could be applied globally over the entire data trace d, by restricting the search space to the segments as determined by the segmented regression, it is possible to search over a much smaller search space. The goal at this point is to find

an automaton with a number of states less than or equal to the number of segments, such that any segment will be covered by one of these states. The dynamic programming of section 3.6 provides an initial segmentation, which is further optimized via this genetic optimization. The key operations of this genetic search are:

- **Genotype** – A sequence of integers of length $|S|$, s.t. each member of this sequence is $\leq |S|$, i.e., for each segment, the index of the state to which it belongs. For all segments that share the same index, D, the dynamics of their shared state is determined via train(m, D)
- **Mutation** – There are two mutation operators, *unify* and *reassign*:
  - *Unify* – Two unique indices are chosen A, B ,and all segments with index A are changed to index B. E.g., $(1, 1, 2, 3), A = 1, B = 2 \rightarrow (2, 2, 2, 3)$
  - *Reassign* – The index assigned to a segment can be changed to any other valid index. E.g., $(1, 1, 2, 3) \rightarrow (1, 4, 2, 3)$
- **Crossover** – Given two index assignments, A, B, randomly choose a cross over index, i, and create children equal to the left half of A concatenated with the right half of B and vice-versa.

  E.g., If $A = (1, 1, 2, 3), B = (1, 4, 4, 4)$ and i is chosen to be 2 then $AB = (1, 1, 4, 4)$ and $BA = (1, 4, 2, 3)$

The *Unify* mutation operator represents an inductive bias acting as a heuristic for the genetic search. The entire purpose of this global merging step is to merge

different segments if their merging results in a more parsimonious model, as such it is likely that merging a large number of segments will speed the search, if their merging is fruitful. For instance, say there are 4 segments representing the same mode, with the *Unify* operator, these will be merged after 3 operations (e.g., $(1,2,3,4) \rightarrow (1,1,3,4) \rightarrow (1,1,3,3) \rightarrow (1,1,1,1)$), with no possibility of backtracking. However, the *Reassign* operator can back track and can easily go back and forth with no fruitful search (e.g., $(1,2,3,4) \rightarrow (1,1,3,4) \rightarrow (1,4,3,4) \rightarrow (1,4,3,3)$) given that each configuration (after the first) will have the same score. As a way of reducing the search time, an initial greedy search is first applied. For all pairs of segments in the optimal segmentation, they are merged if

- They share the same underlying type of dynamics
- The sum of the square distance between their parameters is less than ϵ ($\epsilon = 1e{-}6$ for this work)
- The merging of the two segments results in a lower penalty

The flow of this process can be seen in figure 3.13. The module takes in an entity track, uses dynamic programing to find an optimal segmentation, and then merges modes, first with a greedy merging of the trivially same modes, and then with a global genetic optimization. For instance, in the learning of the y dynamics of the same 1000 frames of *Super Mario Bros.* the initial segmentation has 55 segments. After this initial merging there are 37 segments, with the vast majority of this reduction coming from the merging of the periods where Mario is on the ground, 16 of the 18 merged segments

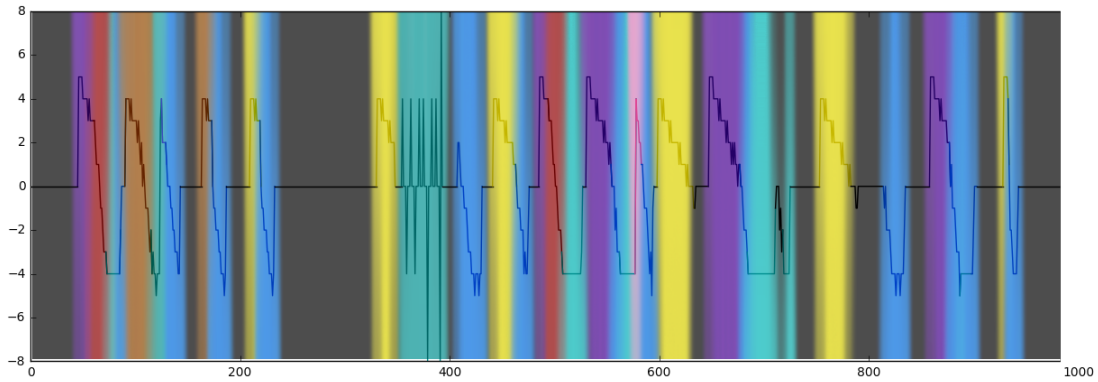


Figure 3.12: One thousand frames of Mario's y-velocity in *Super Mario Bros.*. The color blocks indicate the regions in which *Mappy* believes a mode occurs, with 10 modes being found in total.

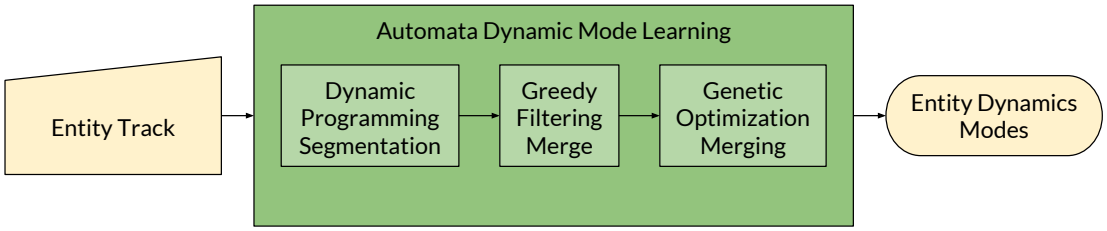


Figure 3.13: The process for segmenting and merging the automata dynamics modes .

88.88%. A model with ten distinct modes is found after 36 and 91 generations with a population size of 100 for MDL and BIC, respectively, the results of which can be seen in figure 3.12. Given the ability to determine *that* a given state is occurring at a given point in time (via the active dynamics or animation), it is now possible to determine *why* a given state was entered or exited. This will provide a crucial step in bringing the automatic annotation up to the standards of the human annotations, if two distinct entities have the same causal effect on a third, it is likely because those two share some similar property, and a reasonable annotation will likely identify those two as being the same in some way (e.g., people annotating the ground and inert bricks as being the same, because they share the same property of "solidity").

3.6.1 Proximate Cause Learning

Given the learned dynamics states from the previous section, a crucial aspect of the automaton remains missing, the causes for transitions between modes. Without this information, *Mappy* would only have a set of dynamics equations, but would lack a functioning automaton. To learn these transitions, first it is necessary to determine what transitions exist in games. Broadly, these transitions can be caused by *exogenous* and *endogenous* factors, i.e.,caused by interactions and sources outside of the state or those from within.

The set of exogenous factors considered are:

- **Collisions** – If the bounding box of an entity's image overlaps with the bounding box of another entity's image, then *Collision Logics* state that it is reasonable

to infer that they are colliding. It is important to note that the direction of a collision can potentially be very important (e.g., Mario colliding with a Goomba from the top will cause the Goomba to die, while colliding from the side will cause Mario to die). A potential source of error in this inference is that it is common for games to have hit boxes that are not the same size as the image presented to the player.

- **Player Input** – Given that interactivity is a hallmark aspect of games (without any response to a player, games are just oddly rebranded movies), player input – specifically, changes in player input – is a crucial source to consider when determining changes in state.

While the exogenous factors are the most obvious causes for a transition, there are a number of endogenous factors to consider:

- **Time in State** – It is common in games for a state to last some specific duration. Most one-shot animations in games are this way, as the entity will enter into a state (e.g., Simon Belmont whipping in *Castlevania*) and that state will exit when the animation has finished. It is important to note that many of these states can be interrupted via the exogenous factors above (e.g., Simon Belmont can be knocked out of the whipping animation if an enemy collides with him). As such, it is important to consider the duration of a state only if no exogenous factors can be linked to the transition.

- **Variable Threshold Crossings** – Similar to the duration of a state, it is possible

for a transition to occur when a given state variable crosses a specific threshold. E.g., when Mario reaches the apex of his jump in *Super Mario Bros.* the gravity value will change, or when Mario's y velocity goes too low then his y velocity is set to a terminal value.

While the endogenous factors are real, it is important to consider them in a secondary manner to the exogenous factors as there can be something of a chicken-or-egg problem. Consider the state change from Mario in terminal velocity to landing on the ground: Mario's $y - velocity$ is -4 while in the terminal velocity fall and then is set to 0. The **Variable Threshold Crossing** endogenous factor would state the purported reason for this transition as "Mario's y-velocity went above $x, -4 < x \leq 4$, causing his y-velocity to change to 0." The condition is true, but is not the causal reason for the transition. In this case, the collision with a solid entity, i.e.,the ground, caused the transition to the standing on ground state. Of course, as mentioned above, this reasoning *can* be correct, as in the case of the terminal velocity – "Mario's velocity went below -4, causing his y-velocity to be set to -4," which, while seeming like circular logic, is actually true causation. Given these classes of events as potential causes for transitions, the goal is to develop machinery for determining which events are the causes. A common measure used in learning causal transitions is that of Pointwise Mutual Information (PMI) [36, 37]. Pointwise Mutual Information is an information theoretic measure that seeks to answer how related two events are. Given two events x, y that come from random variables X, Y respectively, the pointwise mutual information

measures how much their joint distribution differs from their individual distributions. Mathematically:

$$\text{pmi}(x;y) \equiv \log \frac{p(x,y)}{p(x)p(y)} = \log \frac{p(x|y)}{p(x)} = \log \frac{p(y|x)}{p(y)}.$$

Given the identity of independent random variables: $p(x,y) = p(x)p(y)$ iff $X \perp\!\!\!\perp Y$, it is trivial to see that pmi is 0 when two events are independent, is high when two events are perfectly correlated, and is low when two events are perfectly anti-correlated. The high and low ends depend on the probability of the different events, so Normalized Pointwise Mutual Information (NPMI) was introduced to normalize to a fixed scale.

$$\text{npmi}(x;y) = \frac{\text{pmi}(x;y)}{\log p(x,y)}$$

Whereby npmi is again 0 when two events are independent, but is 1 when two events are perfectly correlated and -1 when perfectly anti-correlated. Now, npmi is only a measure of correlation, and, as has been pointed out since the introduction of correlation [143], correlation does not imply causation. However, if the model meets the Causal Markov Assumption [78] certain causal properties will be upheld. The Causal Markov Assumption is:

> A variable X is independent of every other variable (except X's effects) conditional on all of its direct causes. [162]

A model need not capture all possible causes for an event, but all random variables considered in the model must have their causal links fully captured for the model to fit the Causal Markov Assumption. In the case of *Mappy* the model is defined as:

$$\text{collision} \Rightarrow \text{change}, \text{input} \Rightarrow \text{change}, \text{timer} \Rightarrow \text{change}, \text{threshold} \Rightarrow \text{change}$$

which, to first order, upholds the Causal Markov Assumption (collision, input, and change – meaning change in mode – are all events). Conditioned on the state change, it is reasonable to believe that any of the factors listed above are independent of each other. Considering those factors:

- **Collisions → Player Input** – Perhaps the trickiest to disentangle. At first order, a collision in the game is obviously not a causation of the player's input; however, a collision in the game might be closely linked to a change in the player's input (e.g., to use the spring board in *Super Mario Bros.* the player must press A upon landing on the spring board, or a novice player might press jump instantly upon a Goomba, in the mistaken belief that they can affect a change in the game). Certainly, at some colloquial level, the collision "causes" the player to make that change, but there is nothing in the game that necessitates this reaction on the player's part. The player, being an individual with agency, can choose not to press the button. It might be suboptimal, but, nonetheless, remains a choice available to the player.

- **Collisions → Time in State** – Trivial to determine that the collision of two entities will have no effect on the time in the current state passing a threshold.

- **Collisions → Variable Threshold Crossings** – Pertaining to the above discussion of handling endogenous factors as a secondary consideration, the *perceived* threshold crossing occurs as a result of the transition into a new state via the collision. Hence, a variable threshold crossing can not be impacted by a collision,

as the threshold crossing occurs while in a state, and while the collision can affect whether that state is active, it can not affect the actual threshold crossing. (e.g., going from the terminal velocity state in *Super Mario Bros.* to landing on the ground causes a change in the velocity, which naïvely could be seen as a simultaneous threshold crossing, but the velocity change comes as a result of the state change).

- **Player Input → Collisions** – While the player's input will certainly have an effect on the occurrence of collisions, the actual events are independent. Pressing input does not magically force a collision between two entities. In some game states, it might result in a collision, but in others, it might not, and as such, the two should be sufficiently independent.

- **Player Input → Time in State** – Trivial to determine that the player pressing a button will have no effect on the time in the current state passing a threshold.

- **Player Input → Variable Threshold Crossings** – Pertaining to the above discussion of handling endogenous factors as a secondary consideration, the *perceived* threshold crossing occurs as a result of the transition into a new state via the button press. Hence, a variable threshold crossing can not be impacted by the button press, as the threshold crossing occurs while in a state, and while the button press can affect whether that state is active, it can not affect the actual threshold crossing. (e.g., pressing the A button while standing on the ground in *Super Mario Bros.* results in a change in the velocity, but the change in velocity

is a result of the state transition, not a cause of it).

- **Time in State → Collisions** – Trivial to determine that the time in the state passing a threshold will not result in the presence of a collision.
- **Time in State → Player Input** – Trivial to determine that the time in the state passing a threshold will not result in a change in player input.
- **Time in State → Variable Threshold Crossings** – Trivial to determine that the time in the state passing a threshold will not result in the presence of a different variable crossing a threshold.
- **Variable Threshold Crossings → Collisions** – Trivial to determine that a variable crossing a threshold will not result in the presence of a collision.
- **Variable Threshold Crossings → Player Input** – Trivial to determine that a variable crossing a threshold will not result in a change in player input.
- **Variable Threshold Crossings→ Time in State** – Trivial to determine that a variable crossing a threshold will not result in the time in the state passing a threshold.

Given the acceptance that these factors meet the Causal Markov Assumption, then any learned transitions *should* capture the true causal relationships and not merely correlative associations. The transitions are determined thusly.

For all time points that are segmented to be in the same mode during the approach in section 3.6, capture all potentially causal events that occur, and count

Factor	Transition	NPMI
Collision with sky	Ground → Jump	0.044
Collision with ground	Ground → Jump	0.14
Pressing A Button	**Ground → Jump**	**0.73**
Collision with ground	**Fall → Ground**	**0.45**
Collision with pipe	**Fall → Ground**	**0.45**
Pressing A Button	Fall → Ground	0.35

Table 3.1: The npmi values associated with pairs of factors and transitions in *Super Mario Bros.* The true causal pairs are bolded.

their occurrences, c_x. The probability of a given event, x, occurring is then $p(x) = \frac{c_x}{n}$ where n is the total number of time points during those segments. For each transition, y, out of a given mode, A, into another mode, B, count the number of times that a given event, x, is found at the same time c_{xy}, and count the number of times that transition occurs, c_y. The marginal probability of that transition occurring is then $p(y) = \frac{c_y}{n}$, and the joint probability distribution for an event, x, and a transition, y, is $p(x, y) = \frac{c_{xy}}{n}$. The npmi of a given event and a transition can then be calculated via the above equations.

Given the calculation of the npmi for all pairs of events and transitions, a threshold value must be chosen to decide whether the factor causes the transition. Obviously, only positive values need be considered, as there is no reasonable reconciliation of a probabilistically independent event causing a transition. At the other end, while it would be overly stringent to state an npmi of 1, as that would permit no noise in the gathering of events. A threshold value of 0.4 was chosen empirically on examination of a training set. A table of example pairs of transitions and events and their associated npmi values is shown in table 3.1. It is important to note the differences between dif-

ferent types of factors and their associated npmi values. Pressing the A button causing the transition of Ground →Jump has a very high associated npmi, 0.73, while colliding with the ground or pipe has much lower npmi even though they are both true causes for the transition of Fall → Ground. In large part, this is due to the variety found in the causes. Both pipe and ground (and other solid entities) cause the transition of Fall → Ground, so even though every time they occur it leads to the transition, those collisions are not present during every transition, leading to the lower npmi.

It is possible for multiple transitions to be linked to the same causal factor. This can be due to additional state being required to determine which transition should occur (e.g., Mario's jump in *Super Mario Bros.* can be one of three different jumps depending the magnitude of his x-velocity at the time of the jump) or can be stochastic in nature (e.g., walking on the overworld in *Dragon Quest* or *Final Fantasy* will have a random chance of enemy battle). Furthermore, it is possible that no factor has an npmi above threshold. To determine what to do in these cases, a decision tree is constructed.

Decision trees are another white-box machine learning method that allows the training of intuitive `if-else` control flow logic. The C4.5 algorithm [150] is a training method that chooses which variables to condition on to create the most information theoretically pure children. If at any point a maximum depth is reached or a node in the tree is completely pure, it is set to be a leaf node. *Mappy* utilizes the standard C4.5 algorithm, with the major consideration being the data that is considered for a decision.

Only points that are not able to be deterministically labeled are considered. In the case of multiple transitions occurring during a specific factor (e.g., Mario's jump

to the A button press), the only points considered are those points where that factor are active; however, in the case where no transition could be determined, all points in that state are considered. This is the point in which the endogenous factors are considered, but it is possible that only a subset will be considered. Only endogenous factors that could have changed under the from-state of the transition can be used in the decision making process, e.g., if the from-state had a constant y-velocity, then y-velocity is excluded from the decision making process as there is no way it could have passed a threshold.

While the standard C4.5 process is used, the goal of *Mappy* is still parsimonious models, so the process is augmented with an external criterion. *Mappy* trains multiple decision trees, of varying depths, and chooses the best one, using the afore described BIC and MDL. The likelihood value used for a categorical decision tree is the categorical cross entropy:

$$\mathcal{L}_{cce} = \Sigma_{x \in X} l(x) \log(p(x))$$

i.e., for each data point, x, the probability of the labeled value for x, $l(x)$ (1 if that label is the true label, 0 if not), is multiplied by the log of the probability that classifier chose that label, $p(x)$. Due to the limiting of the trees to a maximum depth, it is possible for the trees to be probabilistic, i.e., if they reach an impure leaf node, they will choose a label based on the frequencies of the labels seen in that impure leaf node. The dimensionality of a decision tree is the number of decision points in the tree, i.e.,the number of non-leaf nodes in the tree.

With the learning of a full automaton, the annotation process occurs by looking

at all collisions. While it is possible for entities to exert influence on one another without collision (e.g., an entity might follow another), collisions are the key mode of influence for the collision logic heavy class of games that *Mappy* is designed to work on. All entities that are capable of inducing the same mode transition in an automata are assigned the same property. Similarly, all entities that do not induce any change on a state are assigned the same property. These properties do not have any labels associated with them, other than the fact that they are uniquely identified.

Induction of a change to the dynamical state of an entity is only one possible result of the interaction of two entities. Two other major results of entity interaction are the instantiation of a new entity (or entities) or the deletion of an entity. As above, all entity collisions are considered as possible causes for one of these effects, and the resulting effects are the instantiation and termination points for all entity tracks found during the course of a trace. Certainly, instantiation and deletion can have other causes (e.g., instantiation due to an entities placement in a room, or spawning when the player presses a button, deleting when it goes off screen, deleting on a fixed timer, etc.), but given that *Mappy* only needs to capture what happens when two entities collide, these can be safely ignored, as these events should be independent from the considered factors.

For all times when an entity track is started or deleted *Mappy* looks for all collisions and thresholds based on npmi to find the causal factors. It is important to note that when *Mappy* considers collisions, it does so at varying levels of granularity. To determine the state that an entity is in, it first finds all entities that seemingly share states, as evidenced by them having the same dynamics states and animation

sequences. These merged entities are treated as a generic *stateless* entity, (e.g., two Goombas in *Super Mario Bros.* would be treated as a stateless entity). Similarly, even within one track, an entity might have different So, when two entities collide, it tracks four separate types of collisions: $entity_{stateless} \times entity_{stateless}$, $entity_{stateless} \times entity_{stateful}$, $entity_{stateful} \times entity_{stateless}$, and $entity_{stateful} \times entity_{stateful}$. Of course considering collisions between classes of entities is more generic, and given the same predictive power, more appealing (e.g., ?-blocks in Mario will change state when Mario collides with them, no matter if Mario is in regular, super, or fire state), but if the consideration as the generic entity harms the predictive power, then the collisions that incorporate knowledge of state will be used (e.g., a brick in Mario will bump when regular Mario hits it from below, but will break when super or fire Mario collide, an important distinction that should not be ignored).

3.6.1.1 The Distinction Between Dynamics and Behavior

The section so far has discussed learning the dynamics and has ignored the (some-times blurry) distinction between dynamics, by which I mean the set of rules governing the motion of an entity, and behavior, the processes governing when an entity would "choose" to enter a mode. For many entities, the distinction between behavior and dynamics (e.g., the Goomba in *Super Mario Bros.* walks in a direction until it bumps into a solid entity and then reverses direction), as the causal reasons for mode changes can be directly linked back to the physics simulation of world – just as they would for things that would very commonly be considered dynamics (e.g., the Goomba

colliding with the ground while falling).

In general, the difference between dynamics and behavior comes to the question "Is there reasoning about the state of the world beyond direct collision with an entity?" *Mappy* currently makes no distinction between dynamics and behavior, but a set of predicates that could be added to *Mappy* that would enable *Mappy* to learn distinctions between *behavior* and *dynamics* would be:

- Distance to Nearest Entity of a Type is Above or Below a Threshold
- Distance to Nearest Entity of a Type Along an Axis is Above or Below a Threshold
- A Global Resource is Above or Below a Threshold

The distance predicates would be able to added to *Mappy* with very little additional machinery, but *Mappy* currently has no mechanism for capturing or dealing with resource values.

Upon successful detection of which collisions cause which effects (entity creation, deletion, and state change), *Mappy* has enough information to provide an annotation of the rooms it observes. In the following section, I will detail how all of the previous sections come together to complete the annotation process.

3.7 *Mappy* Annotation Synthesis

As detailed over the last few sections, *Mappy* tackles a large number of subtasks via a set of different modules. Figure 3.14 shows the full architecture for *Mappy*.

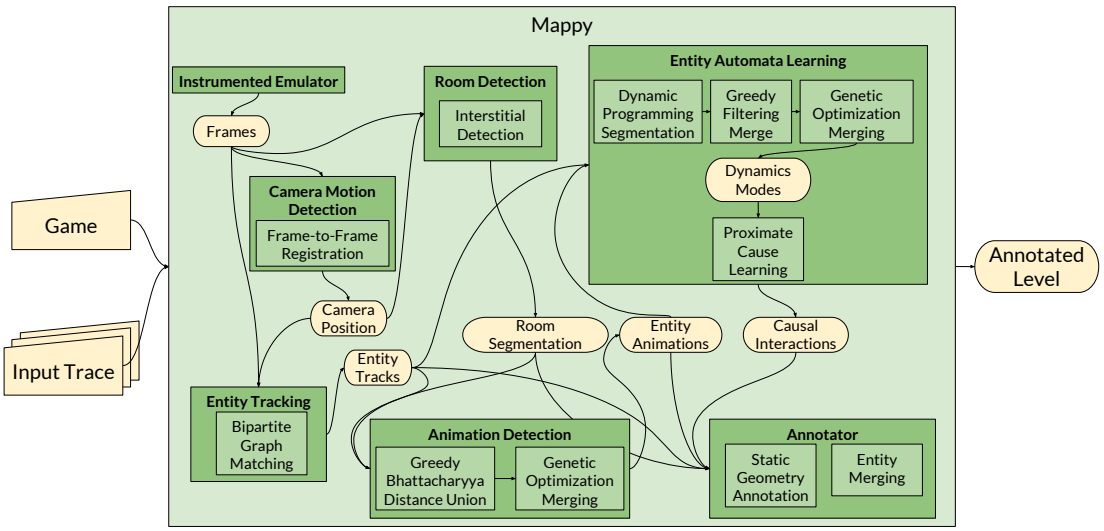


Figure 3.14: The full process for *Mappy*.

Each module relies on a set of inputs that are either provided by a human operator or result from other modules within *Mappy*. The modules are:

- **Instrumented Emulator** – **Input**: *Game Rom*, *Input Traces* – **Output** – Per frame data on all entities (screen position and image displayed) and presence of player control, i.e. *Frames*
- **Camera Motion Detection** – **Input**: *Frames* – **Output** – Per frame information about the camera world position, i.e. *Camera Position*
- **Entity Tracking** – **Input**: *Frames*, *Camera Position* – **Output** – *Entity Tracks*, each with per frame world positions and the image information
- **Room Detection** – **Input**: *Frames*, *Camera Position* – **Output** – *Room Segmentation* of which room is active during which frames, and all entities tracks that begin while in a room have their spawn locations noted

- **Animation Detection** – **Input**: *Entity Tracks* – **Output** – *Entity Animations* describing the animation cycles active for each entity at each point in time
- **Entity Automata Learning** – **Input**: *Entity Tracks*, *Entity Animations* – **Output** – *Causal Interactions* that cluster entities together based on their shared mechanical properties
- **Annotator** – **Input**: *Room Segmentation*, *Entity Tracks*,*Causal Interactions*, *Entity Animations* – **Output** – *Annotated Level* with a graph of rooms, with each entity contained in a room annotated by the set of causal interactions clusters it is present in.

The **Annotator** is simply the collation of all information observed and inferred by *Mappy*. For each room (as determined in the **Room Detection** process described in section 3.4) the observed entities are annotated by causal interaction clusters they are a part of (e.g. a breakable brick in *Super Mario Bros.* is observed to cause a transition in Mario's movement dynamics, as well as its own change in animation when bumped into by Mario). Entities that share animation states and dynamics modes are merged to be considered as one class of entity (i.e. all Goombas are part of the Goomba entity class).

3.8 Conclusion

As detailed, *Mappy*, is a system that seeks to account for the tasks that a human undertakes when observing a game. When a human plays or observes a game, a

number of phenomena occur, and while *Mappy* does not perform these in the same way that a human does, it does attempt to achieve similar outcomes. E.g., while *Mappy* does not have flicker-fusion of rapidly changing images, it does perform tracking that results in a similar outcome – individual images are merged into entities with continuity.

However, there are two major tasks that humans do that are left untouched by *Mappy* – interactive experimentation and common sense reasoning. *Mappy*, as currently designed, utilizes human provided input traces, but when a human plays a game they are interacting with the game in-the-loop. In the future, it would be ideal for *Mappy* to not rely on human input and instead play the game on its own; however, general video game playing is its own field of study [9] that is nowhere near solved. The other capability of humans not met by *Mappy* is that of background knowledge and common sense reasoning. *Mappy* does no reasoning about the symbols it observes – it treats an image as an incomprehensible identifier with no semantics. Obviously, this is not how humans operate. When a human makes a game, they can instantly infer what is background, what is sky, what is ground, what is dangerous, etc. just from the knowledge of the world and game conventions. Just as general game playing is its own field of study, so too is common sense reasoning [46], and, as such, beyond the scope of *Mappy* at this time.

The following chapter will examine existing human annotations for *Super Mario Bros.* and compare them to the annotation produced by *Mappy*.

Chapter 4

Case Study – A Comparison of Human and Machine Annotations for *Super Mario Bros.*

As mentioned in the last chapter, the goal of annotation is to encode, with as little loss as possible, the map of a game, such that salient mechanical information is retained. This is balanced by the goal of reducing the dimensionality of the map-description space, so as to assist a machine-learned generator in the training and generation process. However, there exists no clear way to determine what the optimal encoding is. To determine whether *Mappy* has achieved its task, it is first important to assess human annotations and determine what gaps exist within their coverage of the maps. Given that *Super Mario Bros.* remains the focus of most procedural content generation, it has the widest range of existing annotations. To determine the efficacy of

an annotation, Minimum Description Length will be used. If the mechanical properties of an entity in a room can be uniquely identified, it will have a log-loss of 0 (e.g., the ground and the side of a pipe are both mechanically identical and as such the classification of "Solid" would work for both). If an entity can not be uniquely identified, then it will have a loss equivalent to $\log(p(x|A))$, i.e.,the probability of it being selected, given the annotation (e.g., a Goomba and Koopa are mechanically distinct, so an annotation with "Enemy" would incur a penalty). The dimensionality of the annotation is equal to the number of annotation classes, A. This leads to:

$$\textit{Encoding Penalty} = A(1 + \frac{\log(n)}{2})$$

Where n is the number of individual points being annotated, i.e. the number of tiles present in the levels. Leading to a to:

$$\textit{Mechanical MDL} = -\log(p(x|A)) + \textit{Encoding Penalty}$$

The annotation of Shaker and Abou-Zleikha [166] had the smallest annotation set, consisting of:

- **Ground** – the ground tiles that Mario traverses over
- **Above Ground Solid Inert** – "hills" that are above the level of the ground that Mario needs to jump over
- **Empty** – any empty tile (sky, bushes, etc.) that Mario can freely pass through
- **Enemy** – the presence of any, non-specific enemy
- **Other** – any "special" tile, such as a coin, a breakable brick, or a ?-block

The **Ground** and **Above Ground Solid Inert** are identical mechanically, differing only in the image presented to the player. The **Other** tiles are similar in that they have an effect when Mario collides with them, but no other mechanical binding between them. It is important to note that some tiles are not covered by this annotation, e.g., pipes are not considered under any of the classes. For the analysis, I will consider pipes under the class of *Other*.

Snodgrass and Ontañón's annotation set consists of:

- **Solid Inert** – the ground and hill tiles that Mario traverses over
- **Empty** – any empty tile (sky, bushes, etc.) that Mario can freely pass through
- **Enemy** – the presence of any, non-specific enemy
- **Breakable brick** – any breakable brick tile, regardless of what happens when Mario hits it
- **?-Block** – any ?-Block tile, regardless of what happens when Mario hits it
- **Coin** – a coin that Mario can collect
- **Pipe** – One of the 4 different pipe images

Ignoring the fact that some pipes are mechanically distinct due to the fact that the player can enter them (which no annotation addresses), the **Pipe** and **Solid Inert** annotation classes are mechanically identical. This annotation makes no distinction between the hidden information of what a **Breakable brick** or **?-Block** contain.

Dahlskog et al.'s annotation set consists of:

- **Ground**– the ground tiles that Mario traverses over
- **Solid Inert** – the hill tiles that Mario traverses over
- **Empty** – any empty tile (sky, bushes, etc.) that Mario can freely pass through
- **Goomba** – a Goomba
- **Koopa-Troopa** – a Koopa-Troopa
- **Winged Koopa-Troopa** – a winged Koopa-Troopa
- **Bullet Bill cannon** – the cannon that fires Bullet Bills
- **Bullet Bill support** – the support structure for a Bullet Bill Cannon
- **Breakable brick** – any breakable brick tile, regardless of what happens when Mario hits it
- **?-Block with coin** – a ?-Block tile containing a coin
- **?-Block with power-up** – a ?-Block tile, containing a power-up
- **Coin** – a coin that Mario can collect
- **Pipe, Upper-Right** – The upper-right pipe image
- **Pipe, Upper-Left** – The upper-left pipe image that cluster entities together based on their shared mechanical properties
- **Pipe, Lower-Right** – The lower-right pipe image

- **Pipe, Lower-Left** – The lower-left pipe image

A much more distinct set, with most enemy types being disambiguated, it also disambiguates between the types of entities instantiated.

The human annotation scheme used in my earlier work is:

- **Ground**– the ground tiles that Mario traverses over
- **Solid Inert** – the hill tiles that Mario traverses over
- **Empty** – any empty tile (sky, bushes, etc.) that Mario can freely pass through
- **Enemy** – any enemy
- **Bullet Bill cannon** – the cannon that fires Bullet Bills
- **Bullet Bill support** – the support structure for a Bullet Bill Cannon
- **Breakable brick** – any breakable brick tile, regardless of what happens when Mario hits it
- **?-Block with coin** – a ?-Block tile, containing a coin
- **?-Block with power-up** – a ?-Block tile, containing a power-up
- **Coin** – a coin that Mario can collect
- **Pipe, Upper-Right** – The upper-right pipe image
- **Pipe, Upper-Left** – The upper-left pipe image

- **Pipe, Lower-Right** – The lower-right pipe image
- **Pipe, Lower-Left** – The lower-left pipe image

This scheme differs from that of Dahlskog et al. only in the fact that all enemy types are coalesced into a single category.

I will not detail the annotation scheme of Guzdial and Riedl, but the largest differences come from a much more exhaustive **Empty** set, and from no disambiguation between the types of **?-Blocks**. The cost of annotating *Super Mario Bros.* world 1-1 can be seen in table 4.1. Guzdial and Riedl's minimal annotation scheme has the least gap between MDL and encoding penalty, as the vast bulk of the score comes from the shear size of their annotation set. However, this attention to visual fidelity comes at the cost of having the worst annotation score for mechanical properties, by a large margin For the mechanically focused annotated schemes, the more granular annotation schemes perform better than the more loose schemes.

Figure 4.1 visualizes how the annotation schemes are compared. For each entity that is ambiguously annotated, it is penalized with increasing severity for how rare the correct classification is. E.g., if all ?-blocks are annotated as being the same, then the rarer ?-blocks that contain power-ups have a stronger penalty applied than the more common coin-filled ?-blocks.

There is no one optimal method for annotation, as there will almost always be a tradeoff between mechanical and visual granularity (so long as there is visual variation between mechanically identical entities); however, two major takeaways are

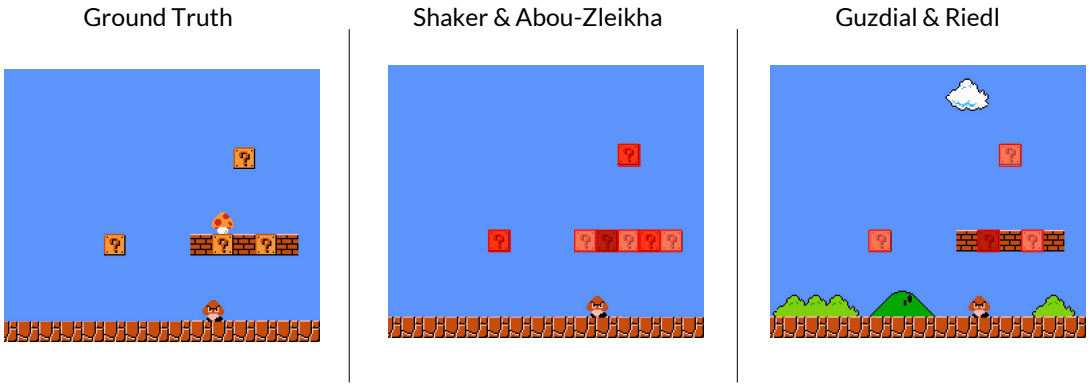


Figure 4.1: A visualization of the mechanical annotation comparison. On the left is the ground truth, and on the right are the annotations of Shaker & Abou-Zleikha and Guzdial & Riedl. Entities that are mislabeled are highlighted in various shades of red, with the darkness of the shade representing the severity of the penalty for the annotation. For Shaker & Abou-Zleikha, the different ?-blocks and breakable bricks are annotated as being the same. This leads to a light penalty for the majority class (the breakable bricks), a moderate penalty for the next most common (the plain ?-blocks), and a large penalty for the rarest class (the ?-block with a mushroom). For Guzdial & Riedl, there is no distinction between the ?-blocks, so there is a light penalty for the majority class (the plain ?-blocks), and a large penalty for the rarest class (the ?-block with a mushroom).

Annotation	Encoding Penalty	Mechanical MDL
Shaker and Abou-Zleikha	**32**	820
Snodgrass and Ontañón	39	123
Summerville (Human)	84	122
Dahlskog et al.	103	**107**
Guzdial and Riedl	791	827

Table 4.1: The MDL cost for annotating *Super Mario Bros.* world 1-1 given the various annotation schemes.

obvious (1) more granularity leads to better visual faithfulness, and (2) it is unknown where the inflection point lies for mechanical granularity. The increase in granularity from Shaker and Abou-Zleikha to Snodgrass and Ontañón to Summerville to Dahlskog et al. leads to a better MDL but with Guzdial, the granularity leads to an increased dimensionality with no additional annotation accuracy. While the annotation of Guzdial and Riedl incurs most of the cost in the encoding penalty, there is a slight penalty due to making no distinctions between mechanically distinct ?-blocks and power-ups hidden in breakable bricks.

4.1 Machine Annotation

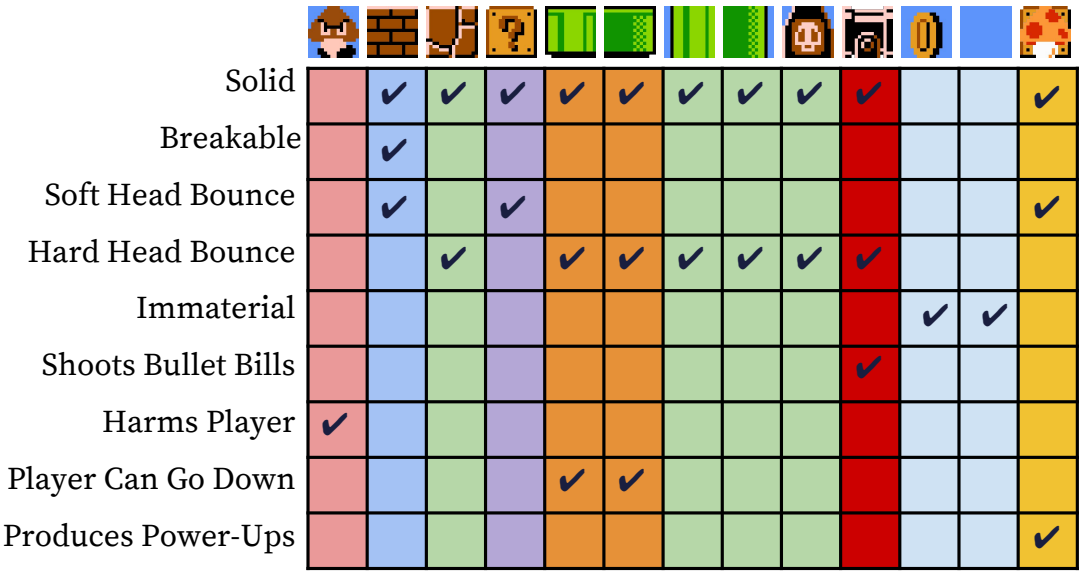


Figure 4.2: Example of the difference between a *Categorical* and *Binomial* annotation schemes for a subset of entities in *Super Mario Bros.*. The properties are listed on the left, and whether an entity has those properties (✔) or not (absence of ✔) , i.e.,a *Binomial* annotation. Alternatively, all columns that share the same properties could be given the same class in a *Categorical* scheme (or each entity could be its own class).

Before assessing *Mappy* and earlier machine annotation methods, it is important to note two distinct methodologies for annotation. All of the human annotations have assumed that the annotations are strictly categorical, i.e.,an entity can have one – and only one – assigned annotation. However, often these individual annotation classes share many similar properties (e.g., in *Super Mario Bros.* the breakable bricks, ground, pipes, and ?-blocks all halt Mario's downward velocity upon contact). An alternative to the *Categorical* lens of annotation is the *Binomial* view – i.e., instead of treating an entity as having one, unique class, it instead has a set of binary indicators for which properties it has. Figure 4.2 shows an example of this distinction. Columns with a unique set of indicators are colored, such that each color would be associated with a single class in a *Categorical* annotation scheme. Alternatively, an entity could be treated as a nonuple of binary indicators in a *Binomial* annotation scheme. Both types of annotation have advantages – in a *Binomial* scheme, the entities can share properties which leads to better generalization; however, entities might be more than the sum of their properties and treating two entities with identical properties as the same might be incorrect (although, this would perhaps indicate that a finer granularity in the properties is necessary).

An early version of this work used an open source clone of *Super Mario World* [146] to gather the data used for the annotation process. This approach utilized the same normalized PMI approach to determine which interactions were most likely to be the causes of effects, as described above, but operated with perfect knowledge of when an interaction occurred and did not need to determine whether a change had even

occurred and if so, what change, as it had full event logs for all state transitions. The approach used a *Binomial* scheme that declared two entities to share a property if they induced the same outcome upon collision with another entity (e.g., the ground and a pipe both halt Mario's y-velocity when he collides with them, so they share a property that might be labeled as "Solidity").

The properties learned by the perfect knowledge method were:

- **Solid** – All solid entities were observed to set Mario's velocity to 0 upon initial contact.
- **Destroyed By Mario Contact From Above** – Coins and Enemies were observed to be destroyed when Mario hit them from above
- **Destroyed By Mario Contact** – Coins were observed to be destroyed upon any contact with Mario
- **Breakable** – Breakable bricks were observed to be destroyed by super Mario upon contact from below
- **Changes To A Spent Inert Block** – The ?-blocks and breakable bricks were observed to be change into a spent inert block when Mario collided with them from below
- **Things That Harm Mario From Above** – The Piranha plants were observed to change Super and Fire Mario's state when collided with from above

In addition, *Mappy* as the end-to-end system described in the last chapter produced two annotations. These two annotations come from two different playthroughs of *Super Mario Bros.*. One of them, *Minimal Interaction*, was run on a trace by a player well-versed in *Super Mario Bros.*. As such, the player knows the mechanics well and ignores most interaction in the room. Entities that act only as diversions, such as the Goombas, are completely bypassed by this player. The second annotation comes from play intended to mimic how a *Novice* would play. The player interacts with nearly of the entities they encounter, and has a nervous, halting style of play (see [202] for a discussion of the types of play shown by novice and familiar players).

The annotation learned from the *Minimal Interaction* trace is:

- **Solid** – Mario transitioned from the jumping animation to the on the ground animation – as well as returned to the on the ground state from one of the jump states – when colliding with the pipes, ground, rocks, ?-blocks, and bricks.
- **Enemies** – The enemies were observed to be the same Hybrid Automaton, an entity that moves at constant velocity to the left (since that is all that was observed in the playthrough)
- **Mario** – An automaton for Mario was learned that covers the three different kinds of jump (standing still, walking, and running), releasing the A button while in one of the jumps leads to an increase in gravity, as does reaching the apex of the jump, and collision with the aforementioned tiles leads to being in the "on the ground" state.

This annotation is, quite obviously, very minimal. Given that very few unique interactions are observed, the only things that *Mappy* is able to extract are that some things move (one, Mario, in a singular way, the rest in a different way) and that some things halt motion.

The annotation learned from the *Novice* trace is:

- **Solid** – Mario transitioned from the jumping animation to the on the ground animation – as well as returned to the on the ground state from one of the jump states – when colliding with the pipes, ground, rocks, ?-blocks, and bricks.
- **Goomba** – The Goombas were observed to be the same Hybrid Automaton, an entity that moves at constant velocity horizontally and changes direction upon collision with other Goombas, Koopas, and solid entities
- **Koopa** – The one observed Koopa was observed to go into a stationary state when Mario collided with it from above and then transition into a very fast movement when collided with in that state (i.e.,when Mario "kicks" the shell)
- **Mario** – An automaton for Mario was learned that covers the three different kinds of jump (standing still, walking, and running), releasing the A button while in one of the jumps leads to an increase in gravity, as does reaching the apex of the jump, and collision with the aforementioned tiles leads to being in the "on the ground" state.
- **?-Blocks** – These were observed to produce coins, mushrooms, or stars when

Mario collides with them from below.

- **Bricks** – These were observed to produce 4 pieces when Mario collides with them from below.

This annotation covers all of the mechanically distinct facets found in the first room of *Super Mario Bros.* (i.e.,World 1-1); however, it is important to note that the automata extracted are not a one-to-one mapping with the reality of *Super Mario Bros.*, to wit, the system never observes Mario in fire Mario state, so it learns no information related to that. This just emphasizes how important a trace (or set of traces) that fully cover all mechanics is to the quality of information extracted by *Mappy*. It is also important to note that while *Mappy* is able to infer that ?-blocks can produce coins, mushrooms, or stars, it does not currently distinguish those blocks that do and those that do not, instead it says there is some probability that a ?-block will produce these. The ability to learn deterministic and stochastic effects is discussed in section 4.3.

The results of *Mappy* on the two discussed play traces and the perfect knowledge approach can be seen in table 4.2. Also shown are the results if *Mappy* took the approach of Guzdial and Riedl – ignoring mechanical similarities and only going off of similar visual representations. The difference between *Mappy* and the relatively exhaustive approach of Guzdial and Riedl is the recognition of animation. Where Guzdial and Riedl include multiple frames of animation that map to separate entities, *Mappy*'s tracking capabilities allow it to utilize a smaller set of annotations (e.g., a Goomba has 2 frames of animation, which allows *Mappy* to devote half the annotation space of Guzdial

Annotation	Encoding Penalty	Mechanical MDL
Perfect Knowledge	71	71
Mappy – Minimal Interaction	12	682
Mappy – Novice	38	75
Mappy – Visual Representation	188	214
Shaker and Abou-Zleikha	32	820
Snodgrass and Ontañón	39	123
Summerville (Human) (MODIFIED)	71	109
Dahlskog et al. (MODIFIED)	84	88
Guzdial and Riedl (MODIFIED)	214	250

Table 4.2: The MDL cost for *Super Mario Bros.* World 1-1 for annotations produced by *Mappy* under a variety of different traces and extraction criteria. The modified annotations are reduced to the set of content only found within World 1-1. Mappy is close to the the MDL of the annotation with no loss of information.

and Riedl's approach).

It is important to note that the cost of the annotation is for a single room, World 1-1, which does unfairly penalize the annotations that cover entities not found in the World 1-1. The human annotations with all content not found in World 1-1 excised can be seen in table 4.2. Dahlskog et al. still have the best annotation score for a human with very similar scores to *Mappy* – Novice, with the key distinction being an increased encoding penalty due to encoding visual distinctions between the ground, pipes, and inert blocks. *Mappy* – Visual Representation and the annotation of Guzdial and Riedl are also very similar, with the key distinction being that *Mappy* – Visual Representation uses 7 fewer distinctions due to its utilization of animations; however, if Guzdial and Riedl were to incorporate Mario's images into their annotation, their annotation space would nearly double – leading to an increase in penalty by 700, whereas *Mappy* would only utilize 1 additional annotation (≈ 6.5 increase to its penalty).

4.2 Case Study – Other *Mappy* Output

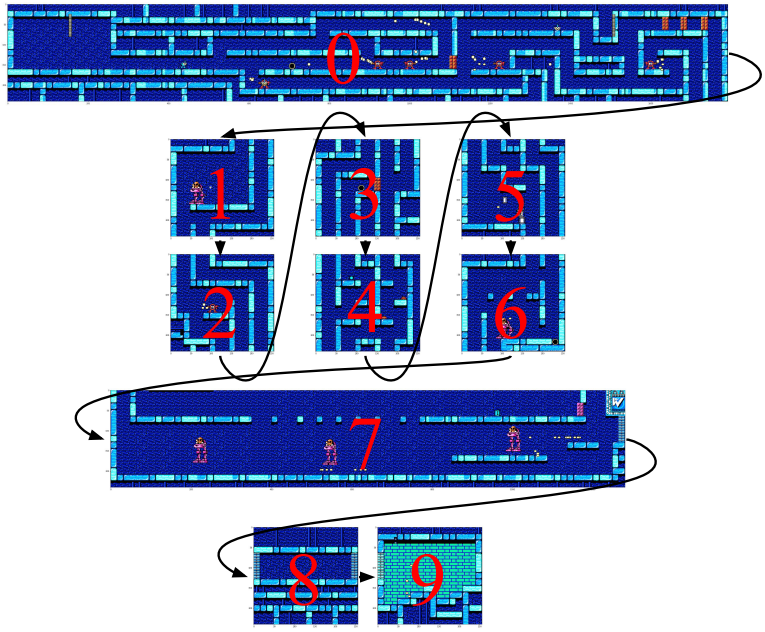


Figure 4.3: The rooms that make up Flash Man's stage in *Mega Man 2*.

Of course, *Mappy* is not just a system for understanding *Super Mario Bros.* and can be applied to other games, with similar results. Figures 4.3, 4.4, and 4.5 show rooms extracted from *Mega Man 2*, *Metroid*, and *The Legend of Zelda*, respectively. All of these demonstrate the tracking, camera motion, and room detection capabilities of *Mappy*, and figure 4.5 demonstrates the room merging capabilities of *Mappy*, whereby rooms that *Mappy* detects as having the same static geometry are clustered together. And, of course, *Mappy* is not simply a system for extracting the visual for rooms, but rather for extracting annotated rooms.

Figure 4.6 shows a play trace through Magnet Man's stage in *Mega Man 3*. Magnet Man's stage has special "magnetic" enemies that pull Mega Man upwards when

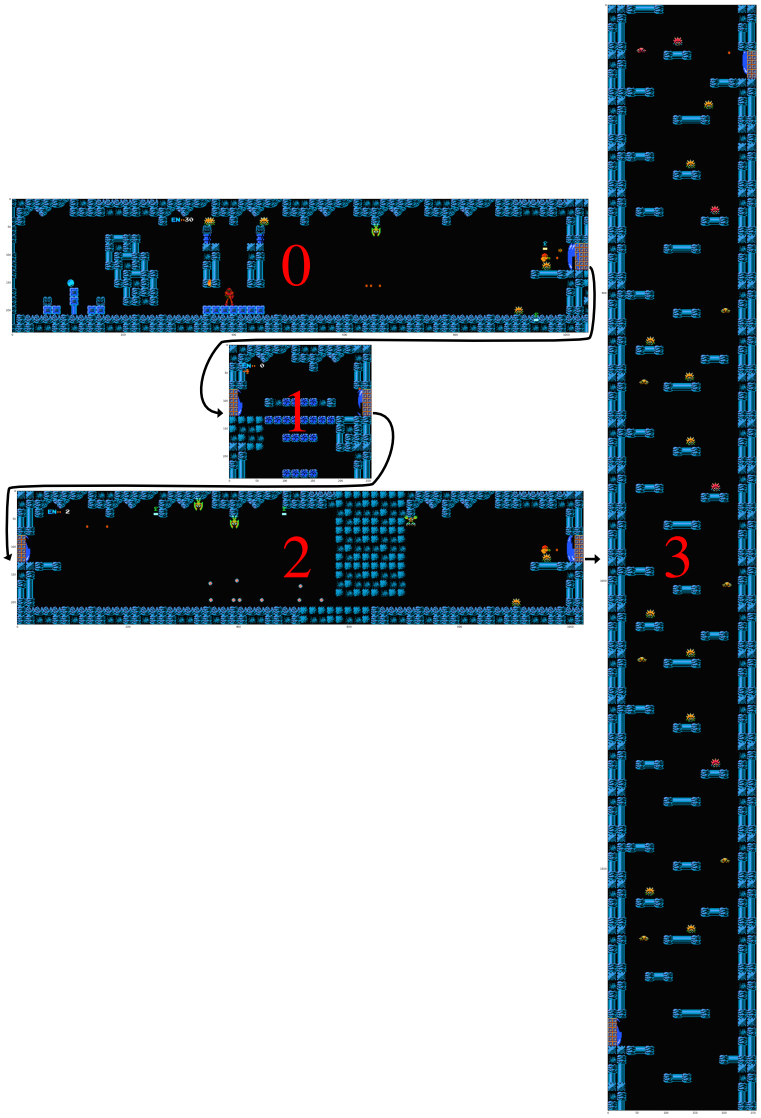


Figure 4.4: The first four rooms encountered in *Metroid*.

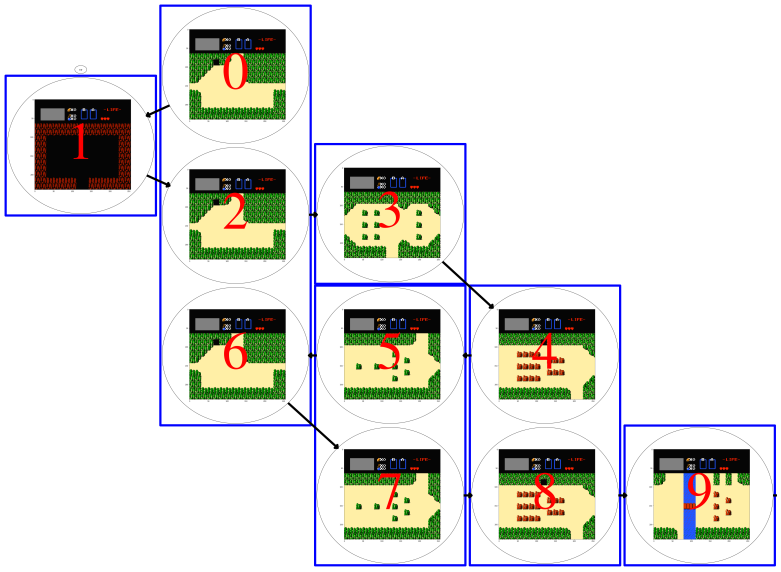


Figure 4.5: Six rooms encountered in *The Legend of Zelda*. The play trace actually sees 9 rooms, but *Mappy* detects that some of these are likely duplicates ([0,2,6], [5,7], and [6,8]).

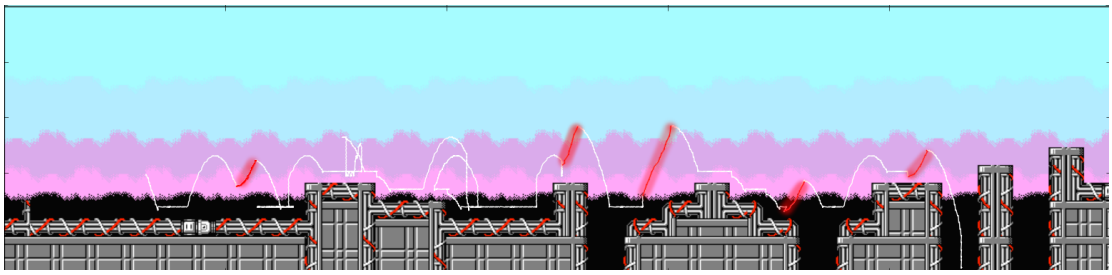


Figure 4.6: Mega Man's track through a section Magnet Man's stage in *Mega Man 3*. The red sections represent places where *Mega Man* was lifted by "magnetic" enemies.

they pass over him. *Mappy* recognizes that this behavior is not like that of normal jumps, since it is not associated with a sudden impulse, but rather a change in gravity; however, since *Mappy* only considers collisions, player input, and state variables, it is unable to determine a cause for these as it does not consider collisions projected on to a single axis. But beyond that unexplained transition, *Mappy* is able to learn all other dynamics for Mega Man, the enemies, and static geometry, learning that the wire covered ground pieces, and the corner ground pieces all have the same effect upon Mega Man – i.e. they are solid.

Considering the automata learned for *Super Mario Bros.* I compare the results to a manually-defined automaton based on human reverse-engineering of the game's program code [93] (see Fig. 4.7). The HAs learned by Mappy can be seen in Figure 4.9. The Mario trace used for this work was 3772 frames in length, 63 seconds. The learned HAs are over-approximations of the true HA. Whereas the true HA has 3 separate jump modes based on the state of $\dot{x}$ at the time of transition, the learned HAs have only one such jump whose parameters are averages of the parameters of the true modes. Following from learning just one jump, Mappy learns only a single falling mode. MDL does learn that releasing the **A** button while ascending leads to a different set of dynamics, but it considers this a change in gravity as opposed to a reset in velocity.

MDL produces the more faithful model of the true behavior, but is overzealous in its merging of the distinct jump mode chains into a single jump mode chain. As such, it only recovers 7 of the 22 modes; however, abstracting away the differences between the jump chains it learns 7 of 8 modes, only missing the distinction between hard bump

On Ground $\dot{y} = 0$ — Caused by Mario colliding with something solid from above

Jump(1,2,3) Three jumps with parameters:

- $\dot{y} := 4, \ddot{y} = -\frac{1}{8}$
- $\dot{y} := 4, \ddot{y} = -\frac{31}{256}$
- $\dot{y} := 5, \ddot{y} = -\frac{5}{32}$

Entered from **On Ground** when the **A button** is pressed and $|\dot{x}| < 1$, $1 \leq |\dot{x}| < 2.5$, or $2.5 < |\dot{x}|$, respectively

Release(1,2,3) $\dot{y} := min(\dot{y}, 3)$ — Entered from the respective **Jump** when the **A** button is released; $\ddot{y}$ same as respective **Jump**.

Fall(1,2,3) Falling at one of three rates: $\ddot{y} = -\frac{7}{16}$, $-\frac{3}{8}$, or $-\frac{9}{16}$; entered from the respective **Jump** or **Release** mode when the apex is reached ($\dot{y} \leq 0$)

Terminal Velocity(1,2,3) $\dot{y} = -4$ - Entered from **Fall** when $\dot{y} \leq -4$. The initial timestep in the **Terminal Velocity** state is actually $\dot{y} = -4 + \dot{y} - \lfloor \dot{y} \rfloor$ before being set to -4.

Bump(1,2,3) $\dot{y} := 0$ — Entered from a **Jump** or **Release** when Mario collides with something hard and solid from below; $\ddot{y}$ same as respective **Jump** or **Release**

SoftBump(1,2,3) $\dot{y} := 1 + \dot{y} - \lfloor \dot{y} \rfloor$ — Entered from a **Jump** or **Release** when Mario collides with something soft and solid from below; $\ddot{y}$ same as respective **Jump** or **Release**

Bounce(1,2,3) $\dot{y} := 4, \ddot{y} := a$ — Entered when Mario collides with an enemy from above; a is given by the respective **Jump**, **Release**, **Fall**, or **Terminal Velocity** state

Figure 4.7: The true HA for Mario's jump in *Super Mario Bros.* := represents the setting of a value on transition into the given mode, while = represents a flow rate while within that mode.

and soft bump. A comparison of the modeled behaviors and the truth can be seen in figure 4.8.

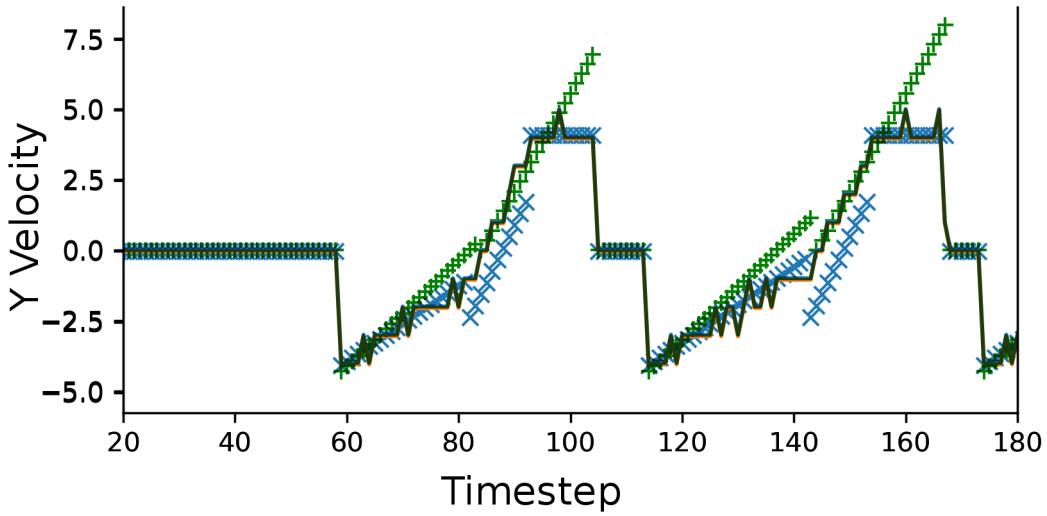


Figure 4.8: Modeled behavior using MDL criterion (Blue X) and BIC (Green +) vs true behavior (Black Line). MDL's largest error source is resetting to an specific value when the true behavior involves clamping to that value, whereas since BIC learns to transition at the 0 crossing it has a more accurate reset velocity. BIC does not learn the transition from **Falling** to **Terminal Velocity**. MDL has a Mean Absolute Error (MAE) of 0.522 while BIC has an MAE of 0.716.

4.3 Discussion

Mappy makes several structuring assumptions about games and play, and it is informative to explore where and how these break down. I have already discussed limitations in scrolling detection and room merging, but there is another important assumption which has not been addressed yet: *Mappy* implicitly assumes it is observing *natural* play where a human explores the game in the way intended by designers and programmers. Here it is useful to distinguish *true* maps from *reasonable* ones. I call maps that accurately reflect game code *true*, even if they are inconsistent with players' expectations of the game's design. A *reasonable* map is one that matches these expectations but might be unfaithful to the source code (e.g. the maps from Zelda El-

On Ground $\dot{y} = 0$ — Caused by Mario colliding with something solid from above

Jump $\dot{y} := [3.97, 4.10], \ddot{y} = [-0.140, -0.131]$ — Entered from **On Ground** when the **A button** is pressed

Release $\dot{y} := [2.10, 2.54], \ddot{y} = [-0.430, -0.384]$ — Entered from **Jump** when the **A button** is released

Fall $\dot{y} := 0, \ddot{y} = [-0.373, -0.359]$ — Entered from **Jump** or **Release** when the apex is reached

Bump $\dot{y} := [-1.85, -1.27], \ddot{y} = [-0.324, -0.238]$ — Entered from **Jump** when something solid is collided with from below

Bounce $\dot{y} := [3.51, 3.82], \ddot{y} = [-0.410, -0.378]$ — Entered from **Jump** when an enemy is collided with from above

Terminal Velocity $\dot{y} = [-4.15, -4.06]$ — Entered from **Jump** or **Fall**

(a) HA with MDL as the penalty.

On Ground $\dot{y} = 0$ — Caused by Mario colliding with something solid from above

Jump $\dot{y} := [4.19, 4.42], \ddot{y} = [-0.195, -0.181]$ — Entered from **On Ground** when the **A button** is pressed

Fall $\dot{y} := 0, \ddot{y} = [-0.356, -0.338]$ — Entered from **Jump** when the apex is reached

Bump $\dot{y} := [-2.37, -1.67], \ddot{y} = [-0.289, -0.188]$ — Entered from **Jump** when something solid is collided with from below

Bounce $\dot{y} := [3.52, 3.88], \ddot{y} = [-0.424, -0.391]$ — Entered from **Jump** when an enemy is collided with from above

Terminal Velocity $\dot{y} = [-4.16, -4.05]$ — Entered from **Jump** when the threshold of -4 is reached.

(b) HA with BIC used as the penalty

Figure 4.9: Learned Mario HAs. Parameters as 95% confidence intervals.

ements mentioned above). Comparing Figs. 4.10 and 4.11 showcases this distinction. The former is a natural play of the game; but in the latter, a tool-assisted speedrunner utilizes multiple glitches to take an optimal (not at all natural) path. The first is the so-called "Up+A" trick, which causes a soft reset of the game when the player enters

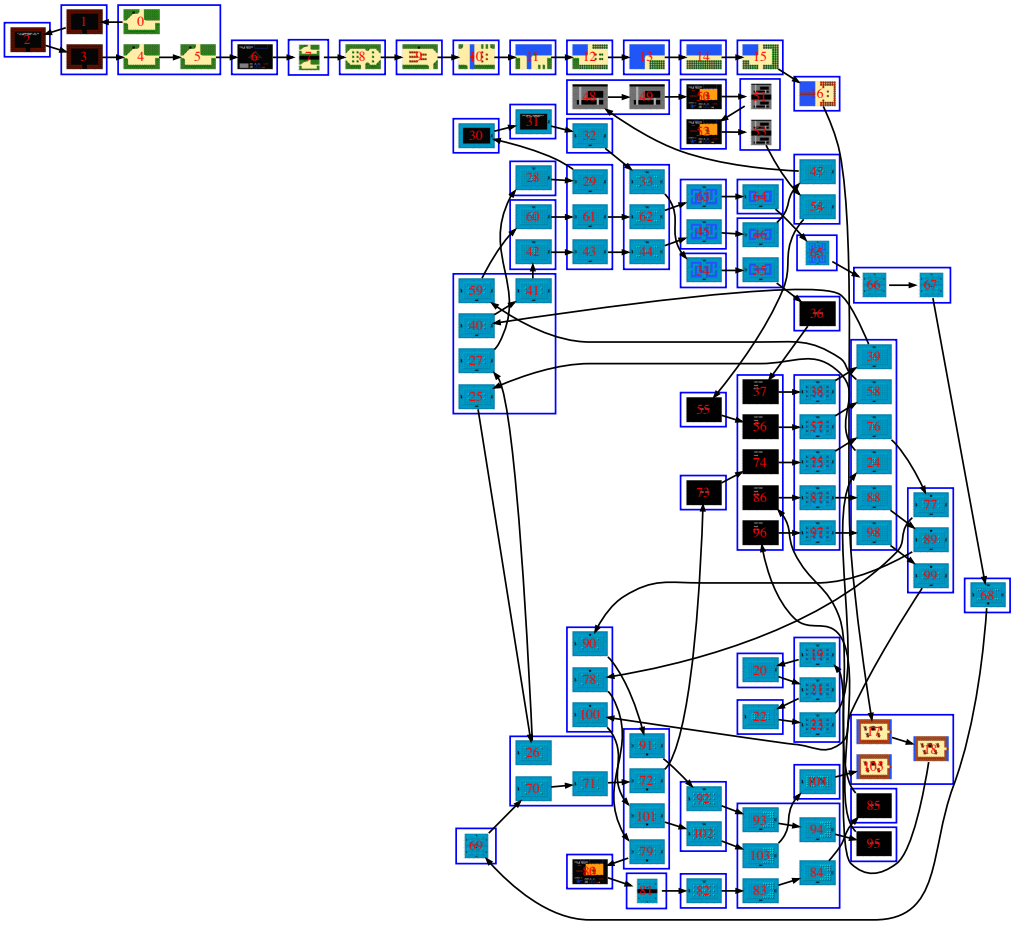


Figure 4.10: *The Legend of Zelda* through the completion of Dungeon 1. The player, myself, made numerous mistakes resulting in deaths (the cluster of black screens in the middle) which teleport the player to the beginning of the dungeon.

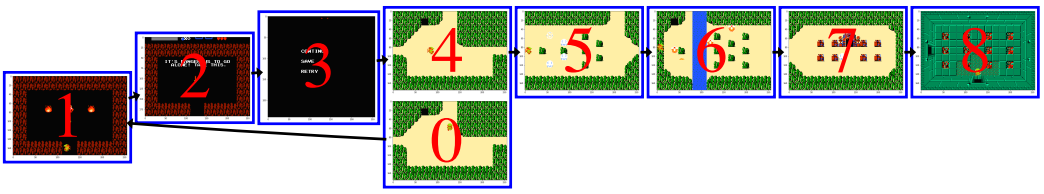


Figure 4.11: *The Legend of Zelda* up to Dungeon 3, showing a map which is *true* but not *reasonable*.

the eponymous command on the second controller. This covers the transitions from 2 (picking up the sword) to 3 (the soft-reset screen) to 4 (the player's initial location at game start). The second trick is "Screen Scrolling," which lets the player leave a screen and re-enter that same screen from the other side. This is how the player warps over the river in 6 and (due to collision detection failing when inside an object) passes back through the same wall to room 7.

All this is allowed by the code of the game, and the *true* map collected captures the full behavior of that code; of course, this would be inappropriate for many of the use-cases. A user feeding this map to a machine learning algorithm for design knowledge extraction would likewise want a map that conveys the intended (if not actual) progressions in the game. It is also interesting to consider that an optimal AI will find such "secret passages" while an AI that does semantic hierarchical planning (e.g. planning sequences of platforms or rooms to traverse) will probably not. That said, *true* maps can be valuable to a game creator—particularly for highlighting areas where it differs from a *reasonable* map e.g. for detecting bugs or sequence breaks.

As for learning map data proper, one important aspect of links which is currently ignored is that links are embedded *in space*. In other words, the player usually travels between rooms because the character stepped on a staircase or crossed between rooms. Right now we do not learn the embedding of the network of links into the tilemaps, but this is important future work. Notably, the same doorway might take the player to multiple different rooms (if, for example, certain game progress flags have been set) or the same room might be entered on the same side from multiple doorways

(as in the Lost Woods in *The Legend of Zelda*).

Another consideration that remains future work is the incorporation of multiple play traces. Currently, *Mappy* only operates on a single play trace at a time, but some facts of a game's map can only be inferred through multiple plays. E.g., in *Super Mario Bros.* the act of going through a warp pipe is destructive – the player will necessarily miss some content, while seeing content not accessible if the player did not go through the pipe – leading to diverging maps that would need to be reconciled through multiple plays. While this is not an intractable problem, image registration and reconstruction is a field of study [31], it remains as future work. Furthermore, multiple play traces would shed light on which effects are stochastic and which are deterministic. E.g., in *Super Mario Bros.* some ?-blocks produce coins when Mario hits them, some produce mushrooms or fire flowers (depending on Mario's state), some produce invincibility stars, and some produce 1-ups. A single play trace of *Super Mario Bros.* does not provide enough information to determine whether these effects are deterministic (hitting the same block will always produce the same effect) or stochastic (there is a $\frac{m}{n}$ chance that any block will produce a given effect). Certainly, there are games where the effects are stochastic, e.g., the power-ups that result from the destruction of an enemy is a pseudo-random effect in *Mega Man*. It is important to note that it is pseudo-random as it depends on a random seed based on the number of frames that the game has been active and the number of visible entities on the screen, meaning that if *Mappy* were to incorporate this information it could feasibly deterministically compute the exact state required to produce a given effect. However, in general this is unlikely to be tractable, as

there are always additional sources of entropy that can be incorporated into the pseudo-random process (e.g., player input, entity positions [including unobserved fractional positions], and non-determinism in memory latency are used in *Final Fantasy*, *Mega Man 2*, and *Legend of Kage*, respectively); instead, *Mappy* will consider stochastic effects as the designers intended, random processes that are beyond the player's manipulation (despite that not being the case).

Another caveat on *Mappy* is that while it handles information from the OAM (i.e.,moving entities or "sprites") and from the nametables (i.e.,static entities or "tiles") in the exact same manner, it does not consider crossover between the two. E.g., in *Super Mario Bros.* a ?-block begins existence as a tile, but when Mario bumps into it, the animation that plays changes the nametable entry to the background color, creates a sprite that moves up and then down (a small bump animation), and finally removes the sprite and changes the nametable to an inert ?-block. To a human, the tile ⇒ sprite ⇒ tile plays out as continuity of one entity, but to *Mappy* this appears to be two separate – albeit tightly interrelated – entities. It is a goal for future *Mappy* to be able to recognize an entity as crossing between tile and sprite, perhaps by noticing that a sprite and tile have (nearly) identical positions and visual representation and both change states at the same time.

Its important to discuss why only World 1-1 was shown, which stems from the relative difficulty in finding good play traces. While there exist many data sources of opportunity [14], these traces typically operate in a way unlike standard human play, stringing together "frame-perfect" tricks (i.e., tricks that require input to come on

exactly the correct $\frac{1}{60}$ of a second) in a way beyond most, if not all, human players. Instead the traces came from a human player, namely myself. It was relatively easy to gather traces for the first room of *Super Mario Bros.*, but it was beyond the scope of the author to produce a trace that fully explores the game and all of its mechanical interactions. Ideally, future versions of *Mappy* would incorporate lessons from General Video Game playing AI (GVG-AI). Instead of taking a game and a play trace, *Mappy* could take a game and learn to explore it [9]. Or, given a game and a trace, use the supplied trace as a starting point for imitation learning [161]. However, this is beyond the scope of *Mappy* at this time.

Part II

The Generation of Game Maps

Chapter 5

Procedural Generation of Two Dimensional Rooms via Machine Learning

Procedural content generation (PCG), the creation of game content through algorithmic means, has become increasingly prominent within both game development and technical games research. It is employed to increase replay value, reduce production cost and effort, to save storage space, or simply as an aesthetic in itself. Academic PCG research addresses these challenges, but also explores how PCG can enable new types of game experiences, including games that can adapt to the player. Researchers also address challenges in computational creativity and ways of increasing our understanding of game design through building formal models [169]. [1]

[1] Portions of this chapter originally appeared in "Procedural Content Generation via Machine Learning (PCGML)" [200]

In the games industry, many applications of PCG are what could be called "constructive" methods, using grammars or noise-based algorithms to create content in a pipeline without evaluation. Many other techniques use either search-based methods [214] (for example using evolutionary algorithms) or solver-based methods [174] to generate content in settings that maximize objectives and/or preserve constraints. What these methods have in common is that the algorithms, parameters, constraints, and objectives that create the content are in general hand-crafted by designers or researchers. While it is common to examine existing game content for inspiration, machine learning methods have far less commonly been used to extract data from existing game content in order to create more content.

Concurrently, there has been an explosion in the use of machine learning to train models based on datasets [128]. In particular, the resurgence of neural networks under the name *deep learning* has precipitated a massive increase in the capabilities and application of methods for learning models from big data [163, 65]. Deep learning has been used for a variety of tasks in machine learning, including the generation of content. For example, *generative adversarial networks* have been applied to generating artifacts such as images, music, and speech [66]. But many other machine learning methods can also be utilized in a generative role, including Markov models, autoencoders, and others [29, 60, 67]. The basic idea is to train a model on instances sampled from some distribution, and then use this model to produce new samples.

Procedural Content Generation via Machine Learning (abbreviated PCGML) is the generation of game content by models that have been trained on existing game

content. The difference from search-based [214] and solver-based [174, 175] PCG is that while the latter approaches might use machine-learned models (e.g. trained neural networks) for content *evaluation*, the content generation happens through search in *content space*; in PCGML, the content is generated *directly* from the model. i.e., the output of a machine-learned model (given inputs that are either drawn from a random distribution or that represent partial or previous game content) is itself interpreted as content, which is not the case in search-based PCG. PCGML can be further differentiated from experience-driven PCG [224] through noting that the learned models are models of game content, not models of player experience, behavior or preference. Similarly, learning-based PCG [154] uses machine learning in several roles, but not for modeling content per se.

PCGML is well-suited for autonomous generation because the input to the system can be examples of representative content specified in the content domain. With search-based PCG using a generate-and-test framework, a programmer must specify an algorithm for generating the content and an evaluation function that can validate the fitness of the new artifact [213]. However, designers must use a different domain (code) from the output they wish to generate. With PCGML, a designer can create a set of representative artifacts in the target domain as a model for the generator, and then the algorithm can generate new content in this style. PCGML avoids the complicated step of experts having to codify their design knowledge and intentions.

Another compelling use case for PCGML is AI-assisted design, where a human designer and an algorithm work together to create content. This approach has previ-

ously been explored with other methods such as constraint satisfaction algorithms and evolutionary algorithms [168, 111, 175].

Again, because the designer can train the machine-learning algorithm by providing examples in the target domain, the designer is "speaking the same language" the algorithm requires for input and output. This has the potential to reduce frustration, user error, user training time, and lower the barrier to entry because a programming language is not required to specify generation or acceptance criteria. If this required a large amount of training data, this could be a terrible task, but in practice a relatively small amount of data is required.

PCGML algorithms are provided with example data, and thus are suited to auto-complete game content that is partially specified by the designer. Within the image domain, there has been work on image *inpainting*, where a neural network is trained to complete images where parts are missing [69]. Similarly, machine learning methods could be trained to complete partial game content.

It is important to note a key difference between game content generation and procedural generation in many other domains: most game content has strict structural constraints to ensure playability. These constraints differ from the structural constraints of text or music because of the need to play games in order to experience them. Where images, sounds, and in many ways also text can be consumed statically, games are dynamic and must be evaluated through interaction that requires non-trivial effort—in Aarseth's terminology, games are *ergodic media* [18].

A map that structurally prevents players from finishing it is not a map level,

even if it's visually attractive; a strategy game map with a strategy-breaking shortcut will not be played even if it has interesting features; a game-breaking card in a collectible card game is merely a curiosity; and so on. Thus, the domain of game content generation poses different challenges from that of other generative domains. Of course, there are many other types of content in other domains which pose different, and in some sense more difficult challenges, such as lifelike and beautiful images or evocative musical pieces; however, I will focus on the challenges posed by game content by virtue of its necessity for interaction.

A number of different approaches have been taken in PCGML, and I will now discuss what is required to generate maps for games in a general sense and the constraints that these requirements place on the representation of a map and the properties of a generator.

5.1 Two Dimensional Room Generation

As mentioned in 3, the focus of this work is on 2D graphical logic based games found on the NES. Within this somewhat restrictive framework, a host of morphologies can be found within the rooms of these games. Some games have a constant room size (e.g., *The Legend of Zelda* or *Donkey Kong*), others have a consistent orientation despite irregular sizes (e.g., all rooms in *Super Mario Bros.* are wider than they are tall), others have irregular orientations (e.g., the rooms in *Metroid* can be much taller than they are wide – $\approx 14:1$ –, much wider than they are tall – $16:1$ –, or roughly equal – $16:15$ –

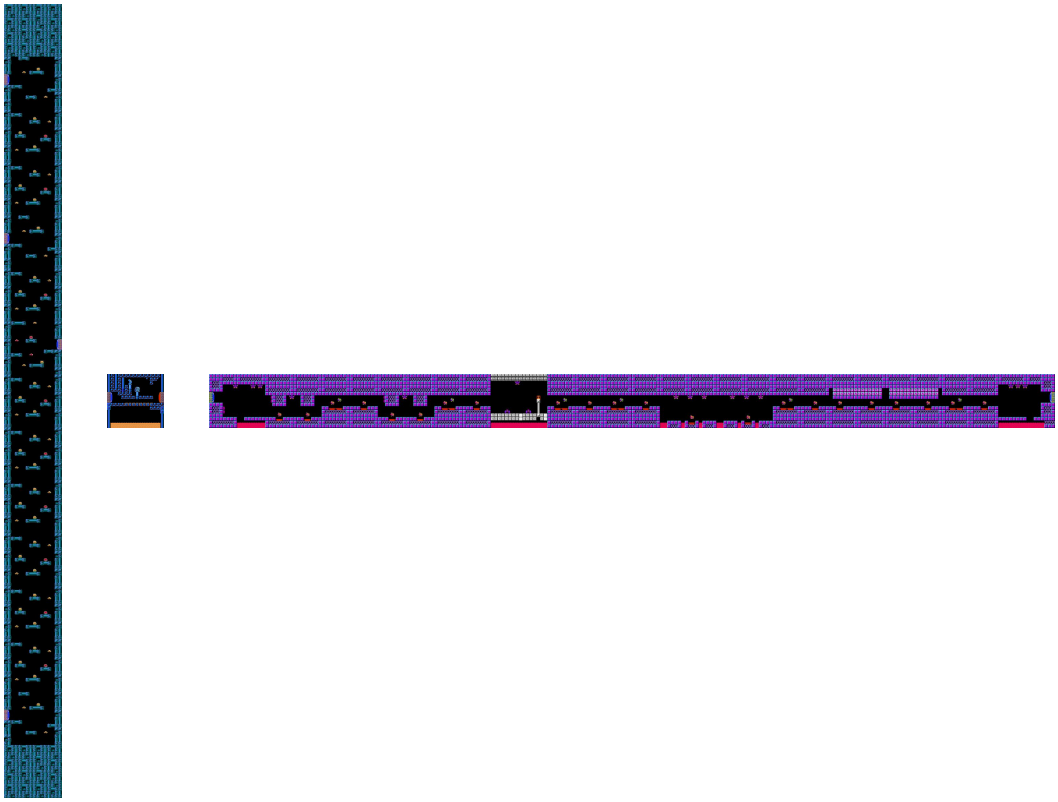


Figure 5.1: The diversity of room size ratios found in *Metroid*.

see figure 5.1), and yet still others have completely irregularly shaped rooms (e.g., the diagonal "auto-scroller" room of *Super Mario Bros. 3* 5-9, see figure 5.2).

Due to the wide diversity in room shapes across games (and within), a general map generator needs to be capable of supporting arbitrary room sizes and shapes. Furthermore, these room sizes and shapes can vary from room-to-room within a map, so no assumptions should be baked into a generator about the size or shape to generate.

To account for the wide variety of room morphologies, a very broad definition of room is required. I define a room as a mapping from a location $(x, y), x, y \in \mathbb{I}$ to the granular pieces of content $\{c_0, c_1, ...\}, c \in \mathbb{N}$ found at that location (if nothing is

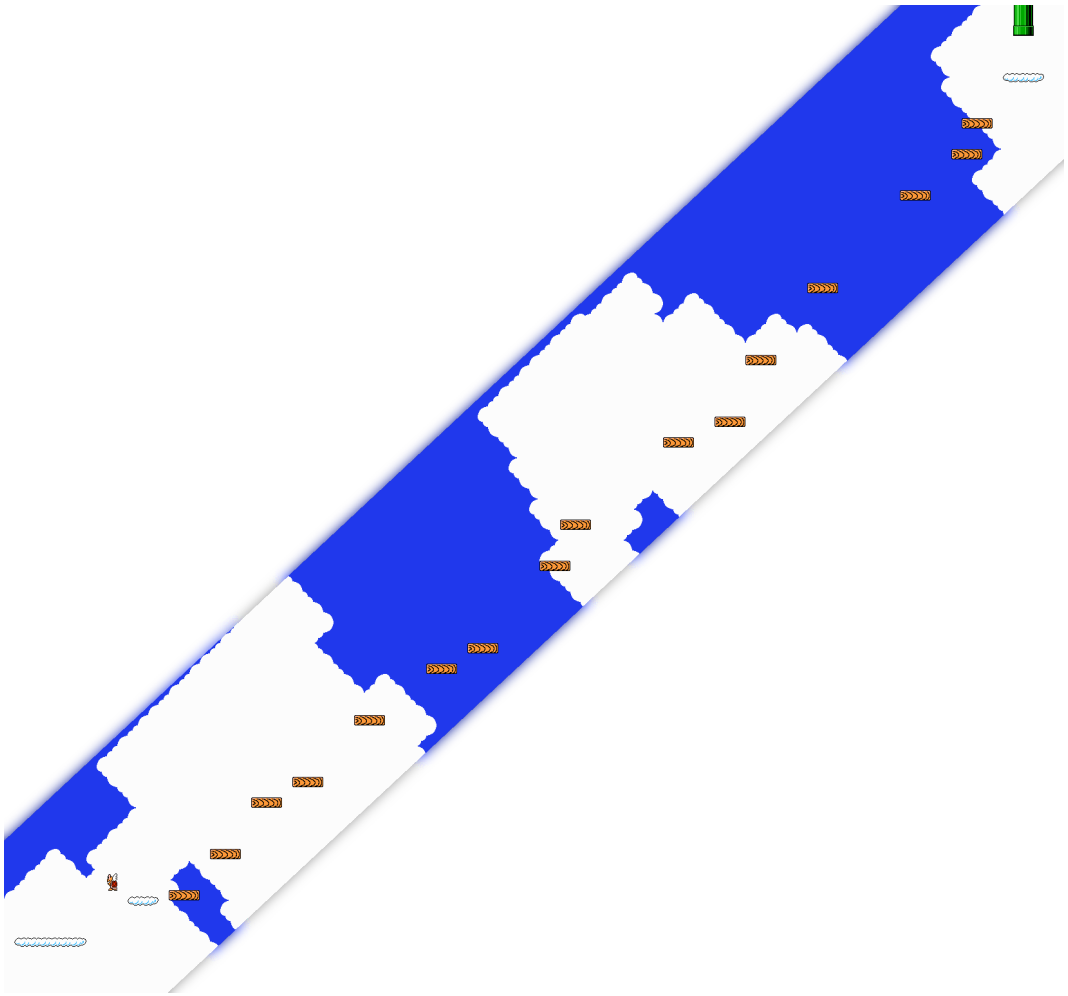


Figure 5.2: The irregular "auto-scroller" room found in *Super Mario Bros. 3*. The camera scrolls at a constant rate diagonally up and to the right, taking the bounds of the room with it. i.e., the room is not a square, as the player will die upon leaving the screen, even if the y-position is higher than a previous location.

at the location then the mapping is to the empty set, $\emptyset$). In practice, most rooms can be represented as rectangular matrices where the entries of the dataset are unique identifiers, as has been done in a large portion of the work [166, 74, 74, 196, 202, 182, 184, 181], but, in general, this is not sufficient to capture the breadth of room content found in games.

This definition allows for a wide variety of different techniques, although some will be better suited to the task than others. Next, I will define qualities that a general room generator must have to achieve the property of generality.

5.2 Required Properties of A General Machine Learned Room Generator

As discussed above, a key tenet driving the use of PCGML is that the generator should learn from the input data, so as to produce content that has the same perceptual and ludic properties as the original dataset. The assessment of how to assess whether a generator has properly learned the design latent within the input, and is able to produce it in its output, is discussed in Chapter 11, but there are other properties that a PCGML room generator must have for it to be capable of generality.

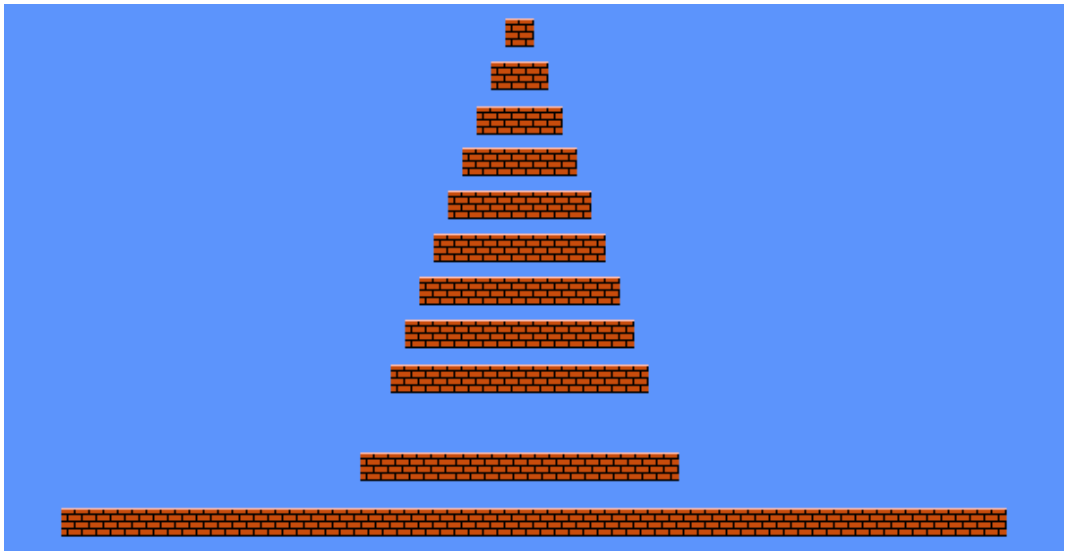


Figure 5.3: The horizontal strips of breakable bricks found in *Super Mario Bros.* The spot for a strip 10 bricks wide is left conspicuously empty.

To examine what is meant by generality, consider the breakable brick in *Super Mario Bros.* Breakable bricks often come in horizontal strips, with sizes of one, two, three, four, five, six, seven, eight, nine, eleven, and thirty-three tiles wide found in the game (see figure 5.3 for a visual); while these are the sizes found in the game, it seems unreasonable to operate under the assumption that it is impossible for a horizontal segment of bricks of length ten to exist. If nine and eleven (and thirty-three for that matter) exist, why not ten? Did Shigeru Miyamoto and Takashi Tezuka intentionally not have a strip ten wide? Or, more likely, was it not a consideration and just did not happen to turn up in the 32 rooms of *Super Mario Bros.*? The goal would be a PCGML system that is capable of generalizing from the input dataset, and capable of producing novel unseen configurations.

To formally define the type of generality considered, we must formally define what a PCGML generator actually *is*. Namely, a PCGML generator is a probability distribution over the possibility space of content. The possibility space contains all possible configurations of the atomic entities making up the content, and a PCGML generator is primarily a system that samples from this space, biasing some regions of more heavily than the others. The areas that the PCGML system is capable of sampling from are the *support* of the generator. Support is a mathematical concept where the support of a function $f()$ is the subset of the domain containing those elements which are not mapped to zero. E.g., a Uniform distribution has equal probability density for all $x \in \{a, b\}$ and is 0 elsewhere, meaning it has support of $\{a, b\}$, whereas the Gaussian distribution has some non-zero probability density for all $x \in \mathbb{R}$ (even though

it approaches 0 as $x \to \pm\infty$). Given this lens, one might define a *general* PCGML system as:

A general PCGML system has infinite support.

Imagine a hypothetical *Copy* generator which takes in rooms from a game map, and when tasked to generate a room, simply randomly selects one of the rooms and returns it. Using this definition of generality, it is easy to see that the *Copy* generator fails miserably, as it will have support of size equivalent to the number of artifacts in its training set.

However, while infinite support is necessary for generality, it is an insufficient criterion. Imagine a hypothetical variant of the *CopyAndFill* generator which takes in rooms from a game map, and when tasked to generate a room, simply randomly selects one of the rooms and returns it after concatenating a random number ($n \in \mathbb{N}$) of some piece of content (say empty background tiles) to the end of the room. This generator has infinite support, but the vast amount of the information is still just being copied from the original input dataset. Infinity is not enough, as generalization to unseen configurations is also necessary for genuine generality.

As such, generality is strengthened to:

The support of a general PCGML system covers all possible configurations of content in the possibility space of its content domain.

Thus, any content that *could* theoretically exist within the possibility space of game *should* be possible to be sampled. This is not to say that all possible pieces of content are likely to be sampled, as some pieces of content should be astronomically

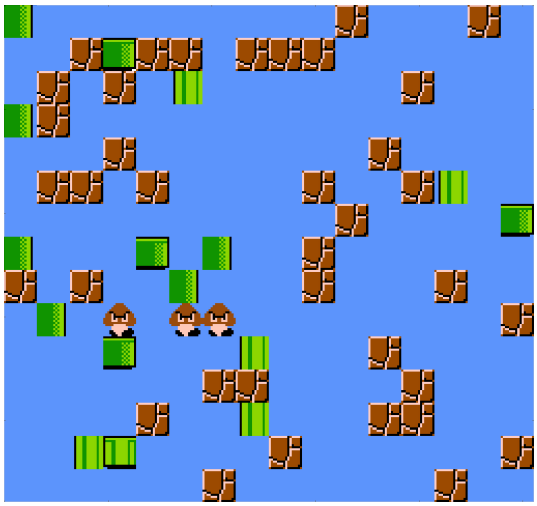


Figure 5.4: A uniformly random *Super Mario Bros.* room.

Figure 5.5: A snippet of a room from *Super Mario Maker.* that plays music as the player runs from left to right. Figure from [210] with permission.

unlikely to exist, given the design latent within the original dataset. E.g., a purely uniformly random room (see figure 5.4) should not be particularly likely, but should still be a possibility. Similarly, other types of rooms, such as music playing auto-

scrollers (see figure 5.5) are possible within the realm of *Super Mario Maker* (a game that allows players to design *Super Mario Bros.* style rooms), and while it is unlikely that a PCGML system trained on the original rooms would ever create such a room in a monkeys-at-typewriter style setting, it should not be a theoretical impossibility.

Of course, there are many novel configurations that are undesirable (e.g., rooms that aren't completable, rooms that are solidly one type, rooms that are uniformly random). So, while a PCGML should be capable of producing any potential output, it should also produce rooms that match the target distribution. i.e., on any given metric (and combination of metrics) the generator should have a similar distribution as the original dataset. However, unlike many search-based PCG [213] approaches, these metrics are unlikely to be targeted via the training process, but rather a good PCGML system should achieve similar distributions as a byproduct of producing high quality content. Determining whether a system has achieved this will be discussed in greater depth in chapter 11. To achieve guarantees of playability (and not just getting by on the luck of sampling well) it is necessary to add on additional constraints (see section 6.6 for more discussion).

In the field of computational creativity, there has been an ongoing discussion about how to evaluate whether a system is creative (and what creativity *is* for that matter). A large part of this debate is whether *process* or *product* is more important for determining the creativity of a system. Colton [41] and Pease [144] argue that both the product and process are important, while Ritchie [153] argues that most evaluations of human creative practice considers only the product, as the inner working of a human

mind is a black box. Colton has responded to Ritchie by citing conceptual art [41], where the concepts and motivations of the artistic process are a key aspect of the art (although this would seem to be an act of converting process into product). Similarly, there is a discussion on P-creativity (the author generates an artifact novel to them) and H-creativity (the author generates an artifact novel to the history of humanity). Conflated with these measures of creativity are that of quality and typicality of the produced artifacts.

Given the larger discussion of computational creativity, why is generality important? This discussion of generality is not meant to decide whether a system is creative or not. Rather, generality is meant to provide an *a priori* way of determining whether the content space supported by one machine learning approach is a strict subset of another approach. If one approach is limited in its expressive range where another is unconstrained, then all else being equal, the latter is preferable to the former. Of course, while an approach might be capable of supporting all possible content in the theoretical limit, it is easily possible that in the practical limit, this is not the case. Generality is not meant to answer that question, but rather to place theoretical limits on the range of content that could possibly be expressed. The evaluation of the content actually expressed – and discussion of what is desired from the expressed range of content – is covered in Chapter 11.

5.3 Existing PCGML Systems and their Generality

5.3.1 Markov Methods

In the context of 2D Room PCGML there have been a number of different approaches. Dahlskog et al. trained n-gram models on the rooms of the original *Super Mario Bros.* game, and used these models to generate new rooms [45]. As n-gram models are fundamentally one-dimensional, these rooms needed to be converted to strings in order for n-grams to be applicable. This was done through dividing the rooms into vertical "slices," where most slices recur many times throughout the room [44]. This representational trick is dependent on there being a large amount of redundancy in the room design, something that is true in many games. Models were trained using various values of n, and it was observed that while $n = 0$ creates essentially random structures and $n = 1$ creates barely playable levels, $n = 2$ and $n = 3$ create rather well-shaped levels. Given that the most granular piece of content that can be placed is limited to vertical slices, if a vertical slice is not found in the original dataset then it simply can not be generated. Similarly, if a pair of tiles do not exist next to each other horizontally, then they can never be generated. Thus, the n-gram approach is not general, unless all possible configurations are found within it.

An extension to the one dimensional Markov chains are Multi-dimensional Markov Chains (MdMCs) [38], wherein the state represents a surrounding neighborhood and not just a single linear dimension. Snodgrass and Ontañón [183] present an approach to room generation using MdMCs. An MdMC differs from a standard Markov chain in

that it allows for dependencies in multiple directions and from multiple states, whereas a standard Markov chain only allows for dependence on the previous state alone. In their work, Snodgrass and Ontañón represent video game rooms as 2-D arrays of tiles representing features in the levels. For example, in *Super Mario Bros.* they use tiles representing the ground, enemies, and *?*-blocks, etc. These tile types are used as the states in the MdMC. That is, the type of the next tile is dependent upon the types of surrounding tiles, given the network structure of the MdMC (i.e., the states that the current state's value depends on). They train an MdMC by building a probability table according to the frequency of the tiles in training data, given the network structure of the MdMC, the set of training rooms, and the set of tile types. A new room is then sampled one tile at a time by probabilistically choosing the next tile based upon the types of the previous tiles and the learned probability table. While the general MdMC approach is more general than the one dimensional approach, it is still limited to whatever is found in the original dataset. However, the approach of Snodgrass and Ontañón falls back to lower-order Markov chains, eventually falling back to strict single entity frequencies, if an unseen configuration is found, which means that the approach has generality.

In addition to their MdMC approach, Snodgrass and Ontañón have explored hierarchical [184] and constrained [185] extensions to MdMCs in order to capture higher level structures and ensure usability of the sampled rooms, respectively. They have also developed a Markov random field approach (MRF) [181] that performed better than the standard MdMC model in *Kid Icarus*, a domain where platform placement is pivotal to

playability; however, unlike the MdMC approach there is no fallback, so a configuration unseen in the input will always be rejected during the sampling process.

Wave-Function Collapse (WFC) by Gumin [70] uses a metaphor borrowed from quantum mechanics wherein there is a "superposition" of tiles used to generate images and rooms from a representative example tile set. This approach is a variant of MRF, except instead of solely sampling, samples are chosen via "collapsing of the wave function" (i.e., probabilistically choosing a tile and propagating constraints that that choice enforces). This in turn can propagate other changes and either deterministically chooses tiles that no longer have any other possible choices or reduces the possible set of other tiles. The probabilities and configurations are determined by finding each $N \times N$ window in the input, and the number of times that window occurs. Given that WFC explicitly precludes the generation of content with a configuration not found in the original dataset, the method itself carries no guarantees about generality.

5.3.2 Neural Methods

Hoover et al. [84] generate rooms for *Super Mario Bros.* by extending a representation called functional scaffolding for musical composition (FSMC) that was originally developed to compose music. The original FSMC evolves musical voices to be played simultaneously with an original human-composed voice [83] via NeuroEvolution of Augmenting Topologies (NEAT) [188]. To extend this musical metaphor and represent *Super Mario Bros.* rooms as functions of time, each room is broken down into a sequence of tile-width columns. Additional voices or types of tiles are then evolved with

ANNs trained on two-thirds of the existing human-authored rooms to predict whether a tile exists within a column, and if it does, at what height. However, this presumes that only one tile of a given a type can be found in each column, meaning that the system is not a general 2D room generator.

In [92] Jain et al. show how autoencoders [218] may be trained to reproduce rooms from the original *Super Mario Bros.* game. The autoencoders are multi-layer perceptrons trained on series of vertical room windows and compress the typical features of Mario rooms into a more general representation. They experimented with the width of the room windows and found that four tiles seems to work best for Mario rooms.

5.3.3 Matrix Factorization Methods

Some approaches to room generation find latent level features in high-dimensional data through matrix factorization, which infers features by compressing existing data into a series of smaller matrices. While often generators create rooms with a limited expressive range [86], Shaker and Abou-Zleikha [166] create *Super Mario Bros.* rooms by first generating thousands with five known, non-ML-based generators (i.e. Notch, Parameterized, Grammatical Evolution, Launchpad, and Hopper). These rooms are then compressed into vectors indicating the content type at each column and transformed into T matrices for each type of content: Platforms, Hills, Gaps, Items, and Enemies. Through a multiplicative update algorithm [108] for non-negative matrix factorization (NNMF), these rooms are factored into T approximate "part matrices" which represent room patterns and T coefficient matrices corresponding to weights for each pattern.

While the "part matrices" can be multiplied by weights to generally generate rooms, the size of the generated rooms is fixed by the dimensions of the part matrices (e.g., in their work, all rooms were 200 tiles wide).

5.4 Graphical Methods

K-means is a clustering method where an a priori defined number of clusters are computed based on the means of the data. Each E step determines which data points belong to which cluster, resulting in new means being calculated for each cluster in the M step. Guzdial and Riedl [73] used a hierarchical clustering approach using K-means clustering with automatic K estimation to train their model. They utilized gameplay video of individuals playing through *Super Mario Bros.* to generate new rooms. They accomplished this by parsing each *Super Mario Bros.* gameplay video frame-by-frame with OpenCV [149] and a fan-authored spritesheet, a collection of each image that appears in the game. Individual parsed frames could then combine to form chunks of room geometry, which served as the input to the model construction process. In total Guzdial and Riedl made use of nine gameplay videos for their work with *Super Mario Bros.*, roughly 4 hours of gameplay in total.

Guzdial and Riedl's model structure was adapted from [97], a graph structure meant to encode styles of shapes and their probabilistic relationships. The shapes in this case refer to collections of identical sprites tiled over space in different configurations. It can be understood as a learned shape grammar, identifying individual shapes and

probabilistic rules on how to combine them. First the chunks of room geometry were clustered to derive styles of room chunks, then the shapes within that chunk were clustered again to derive styles of shapes, and lastly the styles of shapes were clustered to determine how they could be combined to form novel chunks of room. The generation process samples from the style clusters and then samples the cluster to get a specific piece of room geometry (room geometry chunks being contiguous pieces of a given type of game content) and then places those geometry pieces one by one in a scene. However, given that these geometry pieces are taken whole cloth, there is no possibility of the construction of a geometry piece not seen in the original set (precluding something like the 10-wide strip of breakable bricks discussed in Section 5.2).

5.5 PCGML Generality Comparison

In the discussion of generality, there are three classes within which a generator can fall: general, not general, and *dataset dependent*. General and not general are self-explanatory, but *dataset* dependent requires context. Consider the generation task of producing a sequence of tokens, **A** and **B**. If the only observed sequence is **AAABBB**, then a Markov chain of order 1 can only produce sequences that have some number of **A**'s followed by some number of **B**'s – meaning that the generator would not be capable of generating any sequence. However, if the observed sequence was **AABBAB**, then the aforementioned Markov chain would be capable of producing any sequence (assuming the initial token is randomly chosen). This is what is meant by *dataset dependent* – the

Implementation	Technique	Generality
Dahlskog et al.	Markov Chain	dataset dependent
Snodgrass and Ontañón	Markov Chain	**yes**
Snodgrass and Ontañón	Markov Random Field	dataset dependent
Gumin	Wave Function Collapse	dataset dependent
Hoover et al.	Neural Network	no
Jain et al.	Multi-Layer Perceptron	**yes**
Shaker and Abou-Zleikha	Non-Negative Matrix Factorization	no
Guzdial and Riedl	Probabilistic Graphical Model	dataset dependent

Table 5.1: An overview of existing two dimensional machine learned room generators and their generality.

generator is capable of having generality, but only if the dataset contains all possible configurations.

When choosing a PCGML approach to room generation, the theoretical bounds of the generator are merely a starting point, as the actual content produced by the generator is more important for an end user than whether a system could theoretically produce the equivalent of the Shakespeare-from-Monkey's-typewriter with enough sampling, but given the same perceptual quality of output, it is hard to argue for the generator with fewer theoretical capabilities. In the next chapter, I will discuss *Kamek*, a general two-dimensional room generator that utilizes a Long Short-Term Memory Recurrent Neural Network.

Chapter 6

Kamek – A Machine-Learned Two Dimensional Room Generator

As discussed in the previous chapter, there are a number of considerations at play when choosing a technique for training a generator for two-dimensional room generation. The foremost is whether the generator is able to achieve generation of desirable content. This notion of desirability is similar to notions of evaluating computational creativity systems on the *value* and *typicality* of their produced content [153], with the additional caveat that the "creativity" is constrained such that it should produce content similar to that of the original dataset (but not too similar, for what that means – a discussion for Chapter 12). Given the production of high quality content, the generator should have generality (as discussed in the previous chapter). [1]

Given this, the considerations needed by a technique are:

[1] Portions of this chapter originally appeared in "MCMCTS PCG 4 SMB: Monte Carlo Tree Search to Guide Platformer Level Generation"[202] and "Super Mario as a String: Platformer Level Generation via LSTMs" [197]

1. Able to produce any arbitrary local configuration in the room content space

2. Able to produce any arbitrary size and shape of room

3. Capable of capturing long scale dependencies in the content

The technique underpinning *Kamek* is an auto-regressive Long Short-Term Memory Recurrent Neural Network (LSTM RNN). The specific architecture underpinning *Kamek* utilizes a SoftMax output, producing a probability distribution over the atomic entities, allowing it to generate any arbitrary local configuration **(1)**. LSTMs operate on arbitrary sized sequences, allowing *Kamek* to produce output of unbounded size **(2)**. And LSTMs represent the current state of the art for sequence modeling [123], as their ability to hold state in memory allows for long distance sequence modeling [82] **(3)**. Recurrent Neural Networks (RNNs) [85] are a variant of Artificial Neural Network (ANN) that hold state. Where a standard ANN is Markovian (i.e., its output is dependent only on the current input), RNNs can produce different results when given the same input, depending on their state. This non-Markovian behavior is achieved via recurrent edges that reach backwards in time. The mathematical formulation for a "vanilla" ANN (in this case, a Multi-Layer Perceptron with *tanh* activations) is given as:

$$h = tanh(i \cdot \hat{x})$$

$$\hat{y} = o \cdot h$$

Where $\hat{x}$ is the input, $\hat{y}$ is the predicted output, i are the weights for the input-to-hidden layer, and o are the weights for the hidden-to-output layer. A similarly structured RNN is defined as:

$$h_t = tanh(i \cdot \hat{x_t} + r * h_{t-1})$$

$$\hat{y_t} = o \cdot h_t$$

$$h_0 = 0$$

Where r is the recurrent weight connecting the result of the previous hidden step with the current. The key difference between the Multi-Layer Perceptron and the RNN being the hidden activation from the previous timestep found in the RNN, i.e., instead of the input being a single vector, the input is a sequence of vectors. This statefulness allows the Recurrent Neural Network to learn functions that are reliant on their position in a sequence, an important step for capturing longer dependencies.

However, a drawback of the RNN formulation comes from the multiplicative recurrent weight. During training, the error gradient is updated multiplicatively. If $r < 0$, then the error gradient trends towards 0 at an exponential rate during the back-propagation through time (e.g., the error gradient from $h_4 \rightarrow h_1$ is reduced by r^3). This exponential decay in error gradient is known "the vanishing gradient problem." Conversely, for $r > 1$ this gradient exponentially trends to positive infinity, leading to "the exploding gradient problem." Thus, while an RNN is theoretically capable of

learning arbitrary sequence behaviors, in practice the back-propagation through time is truncated to a few timesteps at most, given these problems.

LSTMs are a neural network topology first proposed by Hochreither and Shmidhuber [82] for the purposes of eliminating the vanishing gradient problem. LSTMs work to solve that problem by introducing additional nodes that act as a memory mechanism, telling the network when to remember and when to forget. The, now-standard, formulation of an LSTM is:

$$f_t = \sigma(W_f x_t + U_f h_{t-1})$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1})$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1})$$

$$c_t = f_t \circ c_{t-1} + i_t \circ tanh(W_c x_t + U_c h_{t-1})$$

$$h_t = o_t \circ \sigma(c_t)$$

Where σ is the sigmoid function, $\circ$ is the element-wise multiplication, f_t is the result of the forget gate, i_t is the result of the input gate, o_t is the result of the output gate, c_t is the LSTM cell's state, and h_t is the output of LSTM cell (the W, U matrices are the respective gate's trainable parameters). A graphical depiction of this architecture can be seen in figure 6.1. By disentangling the state of the hidden cell from the output of the cell (the difference between having c_t, h_t in the LSTM and just h_t) allows the LSTM to hold something in memory for a long amount of time without

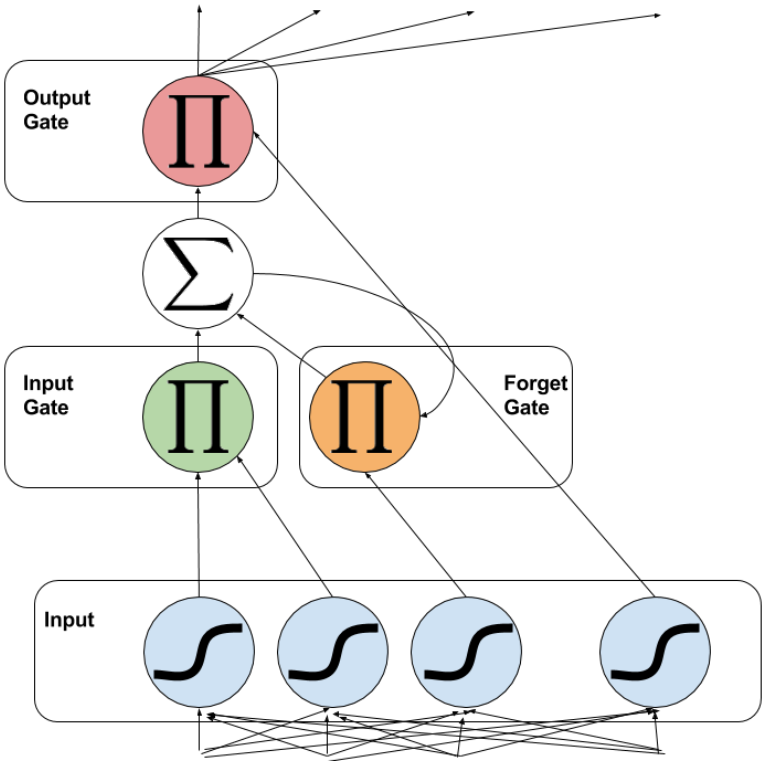


Figure 6.1: A graphical depiction of an LSTM cell.

emitting any external output, leading to the ability to learn behaviors with large time dependencies. Furthermore, the introduction of the forget gate, f_t, [64] allows the LSTM to learn when to decay the gradient through time, meaning that the error gradient need not be propagated past where it is past useful.

While LSTMs are a relatively venerable piece of neural network machinery, they are still the most reliable machine learning approach for sequence modeling. Despite a large number of add-ons to LSTMs [124, 125], and new approaches such as the Recurrent Highway Network [231] and networks found via Neural Achitecture Search [232], the standard vanilla LSTM remains the best neural architecture for sequence

modeling (given a fixed budget for number of trainable parameters) [123].

The LSTMs used in Kamek are trained via the Adam stochastic gradient optimizer and back-propagation. Back-propagation is a method for optimization, most commonly employed in neural networks, wherein a differentiable loss function, in this case Categorical Cross-Entropy:

$$H(p, q) = -\sum_x p(x) \log q(x).$$

where $p(x)$ is a binary 1/0 (1 if that categorical discrete variable was present, else 0) and $q(x)$ is the predicted probability from the LSTM, is minimized via credit assignment to the trainable parameters in the network. The credit assignment process works by taking the partial derivative of the loss, H, with respect to a particular trainable parameter, w.

$$\frac{\partial H}{\partial w} = \frac{\partial H}{\partial o} \frac{\partial o}{\partial i} \frac{\partial i}{\partial w}$$

where o is the output of the neuron that w is a part of, after non-linear activation ϕ, and i is the activation of the neuron before the application of the non-linear activation (i.e., $o = \phi(i)$). Given the partial derivative of the loss with respect to a given weight, the weight is then updated along the direction specified by the partial derivative to minimize the error with respect to itself:

$$w = w - \frac{\partial H}{\partial w}$$

Although, in practice, this update is attenuated by a learning rate, η, as otherwise the gradient tends to oscillate, leading to slow convergence. This learning rate can be adaptive, with a host of different techniques used [52, 212, 102] to accelerate the speed of training. In this work, the RMSProp [212] momentum approach is used. In experiments

[114] it seemed to make little difference which exact momentum approach is used (out of ADAGRAD, Adam, or RMSProp), so it is expected that this is not a particularly impactful choice.

LSTMs are are general neural network mechanism that can be thought to map a sequence, $\{x_0, x_1, x_2, ..., x_n\}$, to an output, $\hat{y}$. In the case of *Kamek*, the LSTM is an auto-regressive model. i.e., it maps from a sequence , $\{x_0, x_1, x_2, ..., x_n\}$, to the next element of the sequence, $\hat{x_{n+1}}$. At the next generation timestep, the predicted element is concatenated onto the sequence, and fed in as the input, resulting in the next prediction $\hat{x_{n+2}}$. This process continues until generation ends.

While LSTMs are the state-of-the art for sequence modeling, the content in question is not a one dimensional sequence. This leads to one of the key contributions of *Kamek* – the handling of two-dimensional rooms as a one-dimensional sequence.

6.1 Data Formulation

As discussed in section 5.1, a room is a mapping of location $(x, y), x, y \in \mathbb{I}$ to the granular pieces of content $\{c_0, c_1, ...\}, c \in \mathbb{N}$. The choice of granular piece of content can be viewed as a Categorical probability distribution. Given that formulation, the goal of *Kamek* is to find the *best* conditional distribution given the context at a generation point, i.e., *Kamek* is finding $Pr(c_{x,y}|c_{a,b}, c_{d,e}, ...)$. *Best* is perhaps overloaded here, but the ideal behavior is such that *Kamek* should produce high quality, novel content that is of the same style as the original input content.

A key question in constructing *Kamek* is

How should a room be linearized, such that the resultant trained generator produces the best content?

Furthermore, *Kamek* must be capable of handling arbitrary room sizes and shapes. While recurrent architectures are almost trivially capable of handling arbitrary lengths of sequences, the addition of grid location requires the generator to keep track of its current location – and also have a way of changing its current location. A key aspect of *Kamek* is the notion of a turtle, a la *Logo* [11]. *Kamek* models a sequence of turtle commands, the result of which construct a reified game room. As such, *Kamek*'s architecture is trying to find the optimal distribution for

$$Pr(c_t|c_{t-1}, c_t{-}2, ...)$$

where $c \in \mathbb{T}$, the set of all commands for the turtle for a possible content space. For instance, in the hand annotated *Super Mario Bros.* content space, the commands available to *Kamek* are:

- $\star$ – Start the generation process
- $\square$ – Stop the generation process
- $\rightarrow$ – Move the turtle to the right
- $\leftarrow$ – Move the turtle to the left
- $\uparrow$ – Move the turtle up
- $\downarrow$ – Move the turtle down

- ↥ – Move the turtle to the highest seen y position
- ↧ – Move the turtle to the lowest seen y position
- ↤ – Move the turtle to the lowest seen x position
- ↦ – Move the turtle to the highest seen x position
-  – Place an empty tile
-  – Place a solid tile
-  – Place a breakable brick tile
-  – Place a coin tile
-  – Place an enemy entity
-  – Place a ?-block that produces a coin upon being hit
-  – Place a ?-block that produces a power-up upon being hit
-  – Place the upper left tile of a pipe
-  – Place the upper right tile of a pipe
-  – Place the lower left tile of a pipe
-  – Place the lower right tile of a pipe
-  – Place the Bullet Bill shooting cannon tile
-  – Place support for a Bullet Bill shooting cannon tile

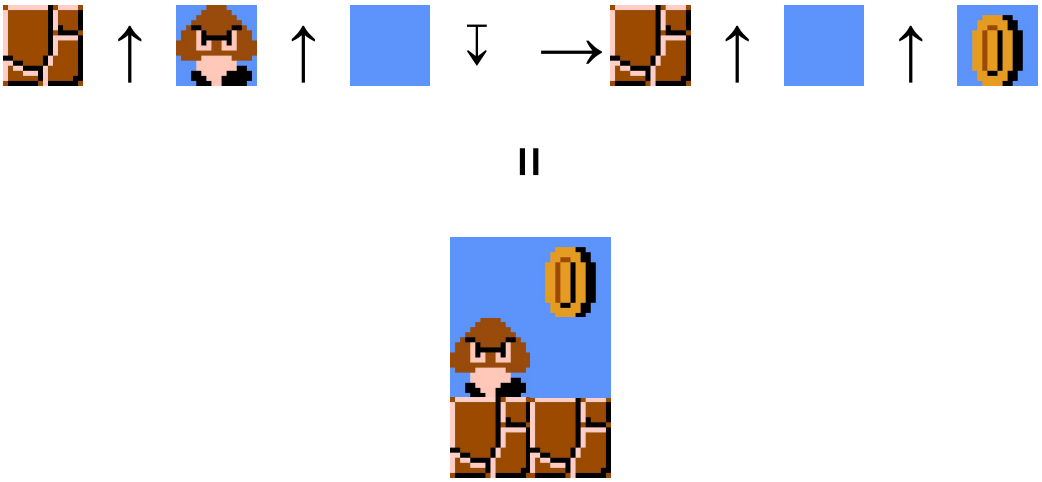


Figure 6.2: An example of the 1-dimensional input stream, and its interpretation when rasterized. Note that the origin is not necessarily the lower-left corner (as it is here), but could be any location as the bounds of the generated room depend only on the places the turtle moves to during the sequence.

An example of a command sequence and the corresponding rasterization can be seen in figure 6.2. The turtle moves during the special commands and then places entities of the given class at generation. In practice, while the turtle can be thought to be controlled via these commands, there are standard patterns that are merged, for the sake of simplifying the generation sequence. These patterns are specific to their specific linearization process and will be discussed in turn for each linearization method.

6.1.1 Sampling Methodology

Given *a* linearization (the specifics of which do not matter too greatly at this point), *Kamek* generates a sequence, which can the be decoded via the turtle and generation commands into a reified room. Remember, the training of *Kamek* is trying to optimize the function $Pr(c_t|c_{t-1}, c_t{-}2, ...)$ such that it minimizes the function:

$$H(t) = -\sum_x (c_t = x) \log(Pr(c_t = x | c_{t-1}, c_t - 2, ...))$$

The result of the training process is an auto-regressive LSTM that produces a probability distribution over the different commands in the generation sequence, which can be sampled from. Starting from an initial input of the start command, $\star$, the LSTM produces a conditional probability distribution, $Pr(c|\star)$. This probability distribution (as with all Categorical distributions), is a vector, p, of real numbers with dimension equal to the dimensionality of the number of possible commands, n, i.e., $p \in \mathbb{R}^n$, such that the sum of the elements is equal to one. Borrowing from simulated annealing [104], this distribution can be sampled at different *temperatures*, $t \in \mathbb{R}$, s.t. $t \geq 0$, changing the shape of the probability distribution. This is achieved by sampling from a new distribution, p', where:

$$p' = e^{\log(p)^t}$$

Three critical values of t to consider are:

- $t = 0$ – Instead of sampling from p' this is equivalent to selecting $\arg\max p$, i.e., taking the index of the largest element of p. This means that the sampling process greedily selects the most likely category from the distribution.
- $t = 1$ – This is sampling from the learned distribution
- $t = \infty$ – This is equivalent to sampling from an uninformative – i.e., uniformly flat – distribution.

The two ranges defined by these critical points are:

- $t \in [0, 1)$ – The sampling will be biased towards more likely results, more deterministic results (being fully deterministic at $t = 0$)
- $t \in (1, \infty)$ – The sampling will be more "random," and further away from the learned distribution

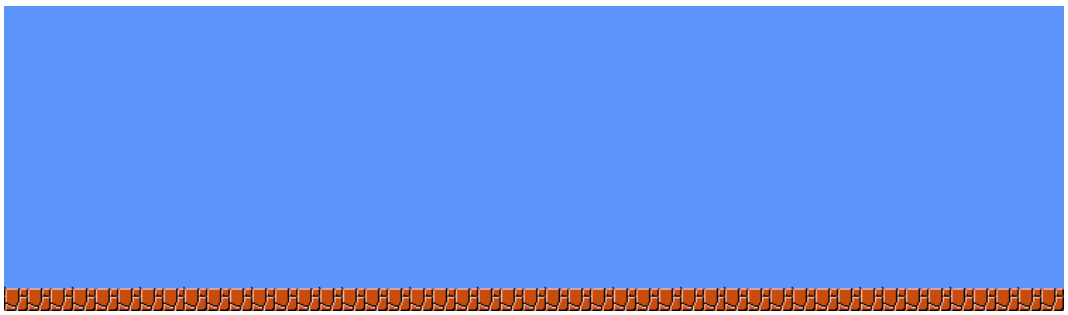
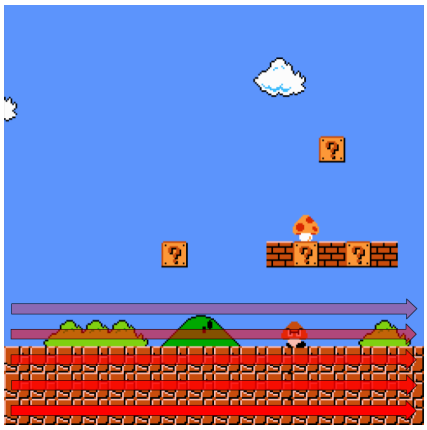


Figure 6.3: The "most likely" room from *Super Mario Bros.* according to an LSTM (three layers, each of 512 units, with 80% dropout between layers). The room is sampled with temperature $t = 0$, i.e., it selects the most likely content piece, greedily. The resultant room is made of the most common column, the ground and empty sky, although the room itself is extremely boring (and funnily enough, highly unlikely).

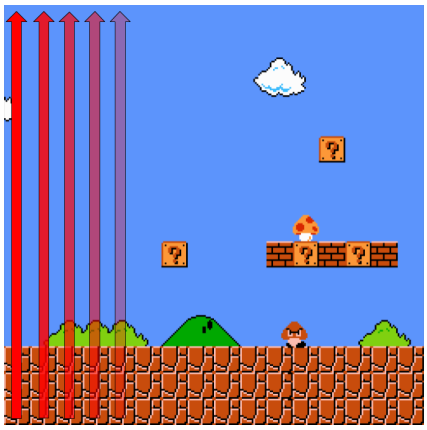
While it might seem that the sampling should be biased towards the most likely, "most correct" values (i.e., a low temperature), in practice this leads to poor results. First, if the temperature were 0, then the sampling process is deterministic, leading to the same content being generated at every step. Second, even if the temperature is non-zero, the most likely results are not actually the best. For an example of why lower temperature does not result in high quality generated content, see figure 6.3.

6.1.2 Choice of Space-Filling Curve

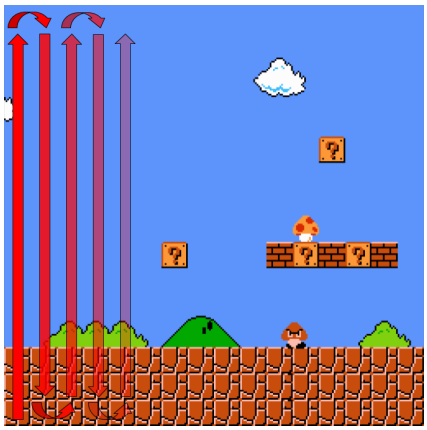
There are limitless possible traversals through the two dimensional rooms, so it is important to understand the properties of the traversals, and how these affect the



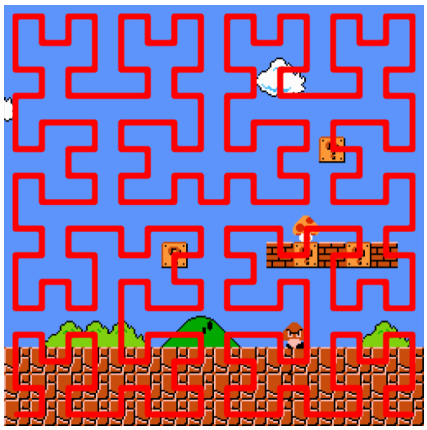
(a) With Grain ordering.



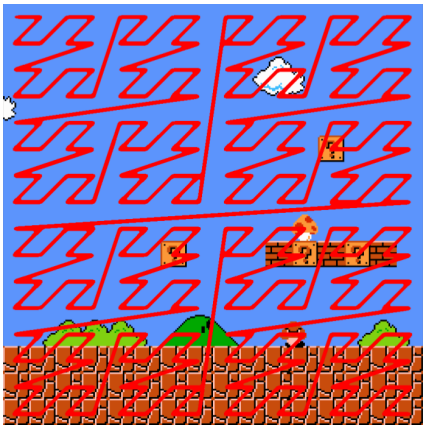
(b) Against Grain ordering.



(c) Snaking ordering.



(d) Hilbert curve ordering.



(e) Lebesgue curve ordering.

Figure 6.4: All orderings considered for this study.

generation process. As a first limitation on the traversals, all curves should have a high degree of locality, i.e., at most points in the traversal, the previous point in the sequence should be near (in a Euclidean *location*-space sense) the current point. In part this is inspired from observations found in statistical machine translation [203], where translation was optimized by reversing the input sequence and then generating an output sequence. The hypothesized reason for reversing the input sequence working better than the forward sequence being that the increased locality (e.g., the first word in the input sentence is very close to the first word in the output sequence) helps the neural network by reducing the duration that it needs to remember.

The simplest, most naïve orderings are that of the *With Grain* (see figure 6.4a) and *Against Grain* ordering (see figure 6.4b) that progresses from either horizontally, then vertically (or vice versa). Given that most game rooms tend to have a specific aspect ratio (e.g. tend to be wider than tall or vice versa), these naïve orderings seem decent. However, as will be discussed, the most naïve assumption – wide room $\implies$ *With Grain* and tall room $\implies$ *Against Grain* is unlikely to be correct. Consider a room in *Super Mario Bros.* which can be up to 400 tiles in length, meaning that a tile might be 400 steps away from the tiles above or below it. Furthermore, even if one of these is correct for a given aspect ratio, a game with inconsistent aspect ratios (e.g., *Metroid*) is going to be forced to use the "incorrect" (or less good at the least) option some not insignificant portion of them time.

The movement patterns found for the naïve orderings are of the kind:

$\star, \mathbb{c}, \rightarrow, \mathbb{c}, \rightarrow, \mathbb{c}, \rightarrow, \mathbb{c}, \rightarrow, \downarrow, \mathbb{c}, \rightarrow, \mathbb{c}, \rightarrow, \mathbb{c}, \rightarrow, \mathbb{c}, \leftarrow\!\shortmid, \downarrow, \ldots, \quad \square$

demonstrating the *With Grain* method (i.e., start, generate, move to the right, generate, move to the right, ..., move down, move all the way back to the left, ...). Given that all generation steps can be safely bundled with a move to the right, these symbols are merged into one during the generation process. Similarly, all instances of moving down can be merged with the left reset, so these symbols are merged without loss of generality. The same are true (albeit with different direction mergings) for the *Against Grain* ordering.

Snaking is a less intuitive manner of progressing through the room (see figure 6.4c). This, unlike the aforementioned curves, has the benefit that the distance in the sequence between a tile and two of its 2D neighbors is always 1 (e.g., consider a *Against Grain* curve on a room 10 tiles high – the top tile is one away from the tile below it and 10 away from its neighbors to the right and left).

The movement pattern for *Snaking* are of the kind:

$\star, \mathbb{c}, \downarrow, \mathbb{c}, \downarrow, \mathbb{c}, \downarrow, \mathbb{c}, \rightarrow, \mathbb{c}, \uparrow, \mathbb{c}, \uparrow, \mathbb{c}, \uparrow, \mathbb{c}, \rightarrow, \mathbb{c}, \downarrow ..., \quad \square$

Similar to the naïve orderings, each generation command has a paired movement command; however, unlike the naïve orderings, the *Snaking* traversal changes directions with each secondary (in this case horizontal) movement. Thus, the merged commands are "Generate and then move in the current direction" and "Move to the right and rotate the current direction by 180 degrees" (the "move to the right" being dependent on the direction of traversal). Unfortunately, this leaves the initial direction unspecified, so an additional symbol must be added to the sequence:

$\star, \Downarrow, \mathbb{c}, \downarrow, \mathbb{c}, ...$

i.e., set the original direction of the turtle to down.

Also considered are two more complex space-filling curves, the *Hilbert curve* (see figure 6.4d) and the *Lebesgue curve* (see figure 6.4e). Both are fractal curves that fill 2-D space. For this work we used a curve with fractal dimension of 4 (i.e., 16×16). For the *Hilbert curve*, when a curve ends (if the origin at bottom-left is $< 0, 0 >$ the end point is $< 15, 0 >$) it progresses into another curve (it begins at $< 16, 0 >$). The *Lebesgue curve*, however, does not tile smoothly left-to-right and instead must be flipped between consecutive curves. For example, if the curve starts in the upper left corner $< 0, 15 >$ it will end in the lower right corner $< 15, 0 >$; a flipped version of the curve then starts at the lower left corner $< 16, 0 >$ and ends in the upper right corner $< 31, 15 >$). Both of these have variable distances between tiles with distances up to 172 (*Hilbert curve*) and 340 (*Lebesgue curve*) which, like the *With Grain*, seem to induce long term-dependencies for minimal payoff. However, unlike the *With Grain* each point in these curves is guaranteed to be within 1 or 2 steps of points in the same columns and rows, as opposed to just the same row. This guarantees that some local structure in the columns and rows should be easily remembered (e.g. that the ground or ceiling is nearby). The large variability in distances also means that *Kamek* must learn the structure of the curve, in addition to the actually pertinent room geometry information. E.g. in the *Against Grain* ordering, the tile 3 away from the top of room meta-character is always 3rd from the top, but in these orderings it could be in nearly any position. These ordering have no common movement patterns, at least, not to the degree that incorporating them reduces the resultant command sequence, and as such, they do not

use them. All of the discussed orderings can be seen in figure 6.4.

6.1.3 Evaluation of Linearization Traversals

Before increasing the parametric search space (network topology, orderings, vocabulary choice), it is important to determine which traversals work well, and under which conditions. The full parametric search space is beyond the scope of this work, taking time and resources beyond what was available, so the search will be performed piece-wise, finding the optimal configuration for specific facets.

Linearization	**Completable %**	**Unique %**
With Grain	100%	34.25%
Against Grain	67.60%	99.95%
Snaking	38.95%	100%
Hilbert	100%	49.80%
Lebesgue	100%	5.40%

Table 6.1: The percentage of the rooms for each linearization that were completable and unique.

To test this, a generator was trained for each linearization approach in the *Super Mario Bros.* domain. The trained networks each consisted of three layers of 512 LSTM cells, with 80% dropout applied between each layer, and were trained using RMSProp [212] until two consecutive epochs found no additional improvement on evaluation of a held out training set. For each approach, the network with the best evaluation score for the held out evaluation set was used for generation.

Each generator produced 2000 rooms and at this stage of evaluation, two proxies for value and novelty are considered, *completability* and *uniqueness*. Completability

is the percentage of generated rooms that are able to be completed by an A^* room playing agent. This approximates the value of the generated rooms, as non-completable rooms are not valuable at all (although a completable room might still fail at other measures of value). Uniqueness approximates P-novelty, i.e., how often a generator produces the same output. The results can be seen in table 6.1. While the *With Grain*, *Hilbert*, and *Lebesgue* approaches each had flawless scores for being completable, this is due to the networks overfitting and memorizing a small subset of rooms, with the worst being the *Lebesgue* approach having only 5.4% of its generated rooms be unique. While, ideally, the generators should produce playable rooms, it is imperative that the generators actually generate unique rooms, else they are complexly trained *Copy* generators (as discussed in section 5.2). Thus, the only linearization approaches kept for further examination are that of the *Against Grain* (with good completability scores, but a slight hit to the number of unique rooms generated) and the *Snaking* (with the worst completability, but producing only unique rooms).

6.2 The Impact of Meta-Information on Quality

While the topological structure of the linearization and the atomic content comprising the room geometry are the only required input and output to *Kamek*, it is possible to imagine a host of meta-information that can help guide the generation process.

For instance, for games where most rooms are large, involved affairs (e.g.,

Figure 6.5: World 7-1 from *Super Mario Bros.* and the relative intensity found throughout it. The room starts (on the left) in a region of calm and then quickly progresses into a challenging section with a lot of cannons for the player to dodge. This is broken up by a small, very difficult section where the player must defeat two "hammer bros." This goes back to the challenging cannon section, which is also followed by another hammer bros. section. The room ends in a relatively relaxed "cool-down" of two pyramids to climb.

Super Mario Bros.), it might be important to consider how "deep" the generator is in the process of generating the room. Figure 6.5 shows how the intensity of a room in *Super Mario Bros.* changes as the player progresses through it. While the LSTM is capable of capturing long-term dependencies, these are long-term relative to other, simpler sequence modeling approaches. i.e., where a Markovian model captures only local dependencies, and an RNN is able to capture dependencies on the scale of 3-6 timesteps, an LSTM can capture dependencies on the scale of hundreds of timesteps; however, in the view of the *Against Grain* or *Snaking* linearizations, 200 timesteps is roughly equivalent to 12 columns of a *Super Mario Bros.* room – not a terribly large amount of real estate. Thus, if it is believed that the progression through a room is important to capture, it stands to reason that the generator might be given a boost to aid this process. To this end, *Depth* meta-information is considered for the generation process.

The *Depth* meta-information is utilized by the LSTM as an additional meta-symbol in the generation vocabulary. For both the *Against Grain* and *Snaking* approaches, there are relatively simple points by which to measure the depth into the

generation process, namely the "Move and rotate the current direction by 180 degrees" merged symbol for the *Snaking* and the "Move with grain and reset against grain" merged symbol for the *Against Grain* approach. Given that it is commonly accepted [98] that LSTMs are capable of capturing temporal dependencies on the scale of 200 timesteps, the depth meta-symbol is incremented every 200 timesteps in the sequence. To give a minimal example of this, assume that the depth meta-symbol, denoted as •, is incremented on every "Move with grain and reset against grain" symbol, ↴, so what was previously:

$$\star, \mathbb{c}, \mathbb{c}, \mathbb{c}, \mathbb{c}, \curvearrowright, \mathbb{c}, \mathbb{c}, \mathbb{c}, \mathbb{c}, \curvearrowright, ..., \quad \square$$

would instead be:

$$\star, \mathbb{c}, \mathbb{c}, \mathbb{c}, \mathbb{c}, \curvearrowright, \bullet, \mathbb{c}, \mathbb{c}, \mathbb{c}, \mathbb{c}, \curvearrowright, \bullet, \bullet, ..., \quad \square$$

Furthermore, while the geometry of the room is what *Kamek* is intended to learn, as described by Byrne, rooms are a "a container for gameplay" [33]. As such, the actions and paths of a player are as paramount as the gross geometry that induce those actions. Toward that end, the path of the player can be incorporated into the generation sequence, via the inclusion of a "Player traversed here" meta-symbol, ☺ .

This player path can be found via one of two – or a combination of both – methods, **(1)** given a forward model for the control of a player (either human defined or automatically extracted via a process like *Mappy* as in chapter 3.6), a goal state (most likely human defined), and some sort of search approach (A^* if a heuristic is defined, Dijkstra's [49] if not) – a simulated set of player's paths can be found or **(2)** actual player paths can be incorporated, if player paths are easily acquired (for instance, acquired

via determining which entity the player controls and utilizing the track information as found by *Mappy* in chapter 3.2). These paths can be incorporated into the generation sequence, such that *Kamek* is forced to generate not just the configuration of game entities that make up a room, but also an exemplar path for a player to traverse that room.

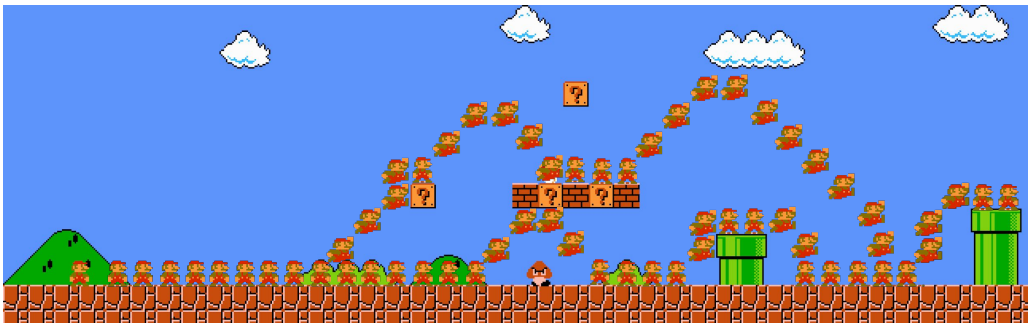


Figure 6.6: An example of two possible player paths through a region of a room in *Super Mario Bros.*

Of course, given either of these methods for getting – perhaps simulated – player paths, there is a question as to *which* paths should be included. The A^* and Dijkstra's will provide optimal paths, but these optimal paths are not necessarily unique. Furthermore, it is possible to find suboptimal paths and incorporate them. Similarly, given actual human created paths, there can be a wide variety of paths available, which might subtly bias the generation process.

An experiment similar to the one in section 6.1.3 was conducted to determine the effect that this meta-information has on the generation process. As before, the trained networks each consisted of three layers of 512 LSTM cells, with 80% dropout applied between each layer, and were trained using RMSProp [212] until two consecutive

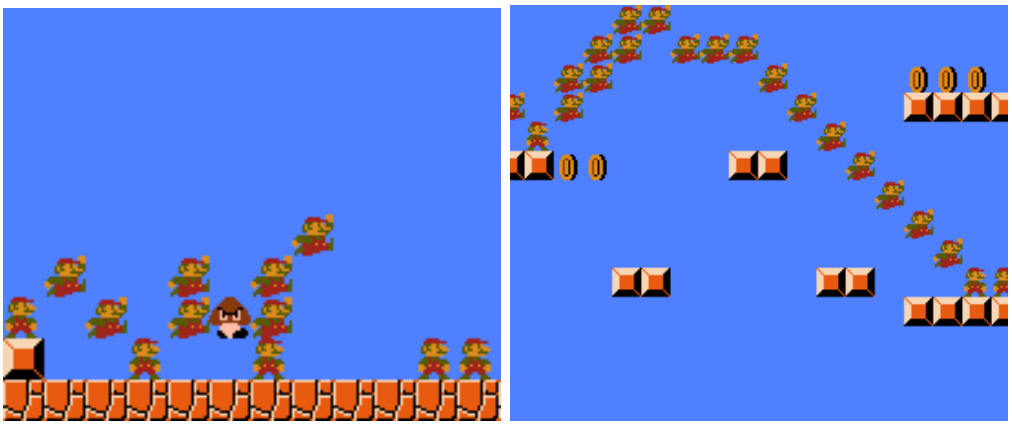
epochs found no additional improvement on evaluation of a held out training set. For each approach, the network with the best evaluation score for the held out evaluation set was used for generation.

Linearization?	**Depth?**	**Path?**	**Completable %**	**Unique %**
Against Grain	N	N	67.60%	99.95%
Against Grain	Y	N	50.65%	100%
Against Grain	N	Y	95.5%	100%
Against Grain	Y	Y	93.25%	100%
Snaking	N	N	38.95%	100%
Snaking	Y	N	39.20%	100%
Snaking	N	Y	92.85%	100%
Snaking	**Y**	**Y**	**97.30%**	**100%**

Table 6.2: The percentage of the rooms for each combination of linearization, depth meta-information, and path meta-information that were completable and unique.

Each generator produced 2000 rooms and at this stage of evaluation, the same two proxies, as used above, for value and novelty are considered, *completability* and *uniqueness*. The path meta-symbol comes from the A^* search used to verify completability; however, instead of taking only the optimal path, all paths within 10 "moves" (i.e., moving one tile space in the room grid) were accepted. While these paths are still very nearly optimal, they do account for multiple paths through the rooms. An example of two such paths can be seen in figure 6.6. This "within 10 of optimal" produces, on average, ≈ 25 paths per room.

The results of the experiment can be seen in table 6.2. The *Depth* meta-information offers very little advantage in completability, minutely increasing the results for the *Snaking* linearization and hurting the completability for the *Against Grain*



(a) A portion where *Kamek* became "confused" and dropped the player's path.

(b) A portion demonstrating an incorrect learned physics "model."

Figure 6.7: Examples of two instance of faulty player paths generated by *Kamek* in the *Super Mario Bros.* domain. The left figure is caused by the sampling process. While it is highly unlikely for *Kamek* to have a large number of timesteps go by without sampling a player traversal symbol, ☺, it is a possible random outcome. The right figure demonstrates an incorrect learned physics model, as the purely diagonal fall is not possible within *Super Mario Bros.*

linearization; however, the proposed reason for incorporation of *Depth* information is to better match the style (this will be examined further in Chapter 11). Perhaps unsurprisingly, the incorporation of player paths greatly increases the completability as *Kamek* is forced to generate an exemplar player path at generation time. While this path can be faulty (see figure 6.7 for two examples), even as an approximation it helps guide *Kamek* to feasible rooms as its generated contented must contain (or given the probabilistic nature of the sampling – will contain at high probability) a path that allows the player to traverse it. Note that the generated paths make no use of a forward model. All "physics" utilized in the generation of these paths are approximations learned by the auto-regressive LSTM.

6.3 Model Capacity

All models generated so far have used a standard architecture of three hidden layers of 512 LSTM units per layer with 80% dropout between layers. However, it is important to assess what size (and dropout) produces the highest quality generated content. Collins et al. [40] found that all common recurrent neural networks (RNN, Gated Recurrent Units, LSTM) have a capacity of approximately 5 bits per trainable parameter. For example, the *Super Mario Bros.* dataset is $2,247,120$ bits of information, for the room geometry. With the incorporation of multiple player paths (the aforementioned ≈ 25 per room), the dataset is $88,364,096$ bits of information. This would seem to indicate that the networks used on this dataset should have somewhere between $\approx 450,000$ and $\approx 18,000,000$ trainable parameters. The number of trainable parameters in an auto-regressive categorical LSTM is given by the equation:

$$\#\text{ of parameters} = 4(nm + n^2 + n) + (n+1) * m + (k-1) * 4(n * n + n^2 + n)$$

Where n is the number of hidden units, m is the dimensionality of the categorical distribution, and k is the number of hidden LSTM layers. For the *Super Mario Bros.* dataset where there are 18 unique elements in the categorical distribution and the assumption of a fixed number of 3 LSTM layers, table 6.3 shows the number of trainable parameters for varying numbers of hidden units per layer.

On the lower end, a network with 128 hidden units per layer is slightly below capacity for the $\approx 450,000$ parameters and on the upper end 1024 hidden units is slightly over capacity for the $\approx 18,000,000$ parameters. This is, quite obviously, a very large

Hidden Units	Trainable Parameters
32	$23,762$
64	$88,466$
128	$340,754$
256	$1,336,850$
512	$5,301,266$
1024	$21,075,986$

Table 6.3: The number of trainable parameters in an LSTM network with categorical dimensionality, $m = 18$, and # of layers, $k = 3$, as a function of network size.

range, and as such it is worth experimenting to find a network size that is well suited to the task. Furthermore, while it is important to find a network that has sufficient capacity to learn adequately from the dataset, it is important to fight the network's tendency to overfit and memorize. Memorization has already been seen in section 6.1.3 where the networks for the *With Grain*, *Hilbert Curve*, and *Lebesgue Curve* linearizations memorized rooms during training. Overfitting is a more general problem, wherein the loss for the training set and hidden validation sets diverge during training – the training loss continues to decrease, while the validation loss hits an inflection point and stops decreasing (and in some instances starts increasing) as training continues. Zaremba et al. [228] found that the application of dropout to LSTMs allowed for the training of larger networks that were better able to generalize to unseen data, than if networks of equivalent size to the dropout were trained. i.e., if dropout of 50% was applied, it performed much better than a network of half the size.

For this experiment, the perplexity of the network on the validation data is used as a proxy for the network to generalize to unseen data. Perplexity, P, is defined as:

$$P = 2^{-\sum_x p(x) \log q(x)}$$

i.e., two raised to the power of the Categorical Cross-Entropy defined in section 6. The intuition behind perplexity is that the perplexity of a fair $k-$sided die has perplexity k. E.g., a fair coin-toss has perplexity 2, and a fair 6-sided die has perplexity 6. Thus, a perplexity of 2 can be interpreted as being a decision where two options are equally likely. However, it is possible for a random variable to have a perplexity of 2 with more than 2 potential outcomes. In that case, the fact that the perplexity is less than the dimensionality means that in some sense, it is an "easier" decision than the uninformative random coin toss. i.e., some outcomes must be more likely and others less, allowing an optimal decision maker some ability to correctly decide the outcome of the random variable.

Holding the decision making task equal – in this case, predicting the next generation command in the sequence – a model with lower perplexity does a better job of correctly capturing the true underlying model – as evidenced by its ability to correctly model unseen data – than one with higher perplexity.

Hidden Units	Dropout	$\lvert 128 \rvert \times$	Train P	Test P	Completable %	Unique %
64	0	≈ 0.3	1.01	1.035	77%	99%
128	0	1	1.003	1.02	98.0	95.5%
256	0	≈ 3.9	1.003	1.011	96.8%	98.7%
256	60%	≈ 1.6	1.003	1.036	94.2%	100%
256	80%	≈ 0.8	1.003	1.038	94.1%	100%
512	0%	≈ 15.5	1.00002	1.005	99.3	18.5%
512	80%	≈ 3.1	1.003	1.012	96.6%	100%

Table 6.4: Perplexity and room quality as it relates to model size and amount of dropout.

The results of the parametric study of network size and dropout can be seen in

table 6.4. To give a point of reference, state of the art perplexity scores on the WikiText-2 corpora (a word level corpus of over 100 million words from Wikipedia)[125]) are ≈ 65.9 which indicates that the prediction of tokens in the *Super Mario Bros.* domain is a much easier problem. This is not surprising, as the most likely room produced by the model (shown in figure 6.3), has just sky and ground. To further frame this problem, a model that predicts this "most likely" room (i.e., sky at the top, ground on the bottom) has a perplexity of 1.11. So, while a model might do a very good job of predicting (due to the abundance of the majority class) it can still do a poor job of generating. Furthermore, while a good test perplexity is important, a large gap between train and test perplexity indicates the model is over-fitting, leading to poor generation performance. As expected, the LSTM with 64 hidden units per layer does not have enough capacity and produces rooms that are not as completable as the other models. While the number of units used at training time is similar between the 128 units with no dropout and the 512 units with 80% dropout, the lack of dropout leads the network to generalize more poorly – leading to non-unique generated content (Using Fisher's exact test they have a statistically significant difference in the amount of unique rooms generated with $p < 0.0001$).

The Birthday Paradox [219, 23] provides a framework for estimating the size of the support of an unknown distribution. The Birthday Paradox is stated generally as "Given a set of n potential outcomes, if we draw k independent and identically distributed outcomes, how many draws do we expect to result in a 50% probability of finding two identical outcomes", or stated in the specific instance "How many people do

you need to gather for a 50% probability that two of them share the same birthday?" This problem is computationally intractable to solve for large sample sizes, but the Taylor approximation for it is :

$$Pr(\text{Collision}) \approx 1 - e^{\frac{-n(n-1)}{2k}}$$

To find an empirical estimate for $Pr(\text{Collision})$ and k, I ran of Monte Carlo trials. For each trial, the output of a generator is randomly selected, without replacement, k times and assessed for whether it contains two rooms that are identical. The estimated $Pr(\text{Collision})$ is then $\frac{\text{Number of Trials with a Collision}}{NumberofTrials}$. For this, I ran 10,000 trials, and found that the 128 unit generator had a k of 67 to reach $\approx 50\%$ – which leads to an estimated size of support of $\approx 3,238$, i.e. it is estimated that the 128 unit generator can produce 3,238 distinct rooms. For the 512 unit generator with 80% dropout, it is impossible to calculate this in the same way, given that no collisions were found. In fact, the generator was ran for 10,000 rooms without producing a repeat. However, a very conservative estimate can be made, if we assume that the 10,001th generated room is identical to one of the previous rooms. This leads to a k of $\approx$ 7,258 to reach $\approx 50\%$ – meaning the support is at least $\approx$ 35 million unique generatable rooms.

We see that increasing dropout is the key factor in this reduction of overfitting – the networks with 256 units see an increase in unique rooms as dropout is introduced, at the cost of reducing the number of completable rooms. In general, we see that increasing network size increases the percentage of valuable – i.e., completable – rooms generated with increased dropout being necessary to combat the tendency to overfit. As such, the general guideline for auto-regressive LSTMs is to use the largest network

possible, with the highest amount of dropout that (1) still has large enough capacity for learning the dataset and (2) is able to converge during training.

A further delve into the evaluation of a room generator, and determining whether it has successfully captured the desired properties of the original dataset *and* learned to generalize is discussed in Chapters 11 and 12.

6.4 On the Importance of Following the Flow of Play

In most of the PCGML work in platformers, generation progresses from left-to-right, an artifact of how most platformers progress – and certainly, for approaches centered on *Super Mario Bros.* all rooms progress from left-to-right. However, there is no reason that generation need progress in that order. Some approaches have operated in a global manner [166, 181], but most have operated in a linear manner progressing from left-to-right [182, 180, 75, 73, 84, 45], including all of the work shown so far. While the global approaches are robust to directional dependence, a question remains as to whether sequence dependent methods depend heavily on the order of generation. The only testing of this came from Snodgrass and Ontañón [183] who tested their system generating from top-to-bottom vs bottom-to-top, finding that for Markov chains, that generation from the bottom was much more preferable, mostly due to the top-to-bottom chains' failure to place the ground in the correct location. LSTMs do not hold that same problem, as the best functioning approach utilizes both top-to-bottom and bottom-to-top generation, but it is necessary to test if the left-to-right dependence is or is not

important. Using the 512 unit, 3 layer, 80% dropout architecture, a generator was trained but with the room being represented from right-to-left, i.e., "backwards." The results can be seen in table 6.5.

Direction	Completable %	Unique %
Left-to-Right	96.6%	100%
Right-to-Left	97%	99%

Table 6.5: room quality as it relates to the direction of generation.

Slightly surprisingly, the right-to-left generation has an impact on the uniqueness, leading to a small amount of duplication in the generative set. Furthermore, the "backwards" direction not only does not hurt the playability of the rooms and in fact produces more playable rooms than the presupposed correct direction (although not statistically different $p = 0.7034$ with Fisher's exact test). This leads to the conclusion that for *Kamek*, the generation order does not matter, an important aspect for extension into games where there is no determinate direction to the flow of play.

6.5 A Comparison of Human and Machine Annotation

Mappy, as detailed in Chapter 3, is a system capable of producing annotations, but the impact of these annotations on *Kamek*'s ability to learn and generate needs to be assessed. The annotation detailed in Chapter 4 is used and the resultant generated rooms are compared with the human annotation. Each generator generated 1000 rooms with the same initial seeding.

The *Mappy* annotation definitely fares worse than the human annotation (with

Annotation	Completable %	Unique %
Human	96.6%	100%
Mappy	94.0%	100%

Table 6.6: Room quality comparison between the author's annotation and that of *Mappy*.

$p < 0.0157$ using Fisher's exact test) for room completability as seen in table 6.6. Oddly, the difference seems to come from the *Mappy* based generator learning an incorrect internal model of the game's physics (e.g., in generation it often generates rooms that require Mario to be able to jump further than he is capable of – and it generates an exemplar path demonstrating this). This internal model should not be affected by the annotation, and in future work, I would like to explore this in more detail. However, the vast majority of rooms generated are of sufficient quality, which is promising for future work in fully removing the human-in-the-loop for this annotate and generate process.

6.6 Combining Machine-Learned and Search-Based Content Generation

While the local-sampling approach underpinning *Kamek* is able to, with regularity, generate rooms that are playable, this process comes with no guarantees. However, if the process of sampling the LSTM (or other probability distribution) is viewed as a random – if heavily biased – generation state transition process, it is possible to supplement this with search-based approaches, of which many exist [213].

However, while there has been research into using search algorithms for content

generation, the determination of *what* the space that is actually being searched is been the critical problem. These approaches have relied on hand authored genetic phenotypes [186], fitting together content chunks [121], or hand-tuned transitions [170] – leading to the searching of a space that has some valid rooms, but the totality of the search space might not be capable of creating all rooms desired, i.e., might lack generality. If the search-space is directly related to the desired content space – and comes from a PCGML system with generality, this searching can combine the strengths of both PCG approaches – the generality and relative ease of defining the search space inherent to ML based approaches and the ability to easily tune global parameters and constraints that come from a search-based approach.

While many search approaches exist, perhaps the easiest pairing for a sampling based ML approach is that of Monte Carlo Tree Search (MCTS). MCTS operates in four steps:

- *Selection* - Starting from the root node, the child with the most potential is chosen successively until a leaf node is found.
- *Expansion* - While that node has untried moves, a move is chosen, at random, and applied
- *Rollout* - Moves are applied randomly until a terminal condition is met
- *Backpropagation* - A score is calculated for the rollout and applied successively to all parent nodes

As mentioned, the sampling based PCGML systems (such as Markov Chains or the auto-regressive LSTM of *Kamek*) start from a seed state and randomly transition to successor states. This process repeats until a terminal condition is reached, be it a terminal state or some desired condition of the chain is met (e.g. length $\geq$ some threshold). From this description, it is easy to see that standard Markov chain generation is directly analogous to a single rollout in MCTS. While MCTS is typically considered in the style of an adversarial game playing AI, hence the parlance of "moves," for our purposes the "moves" that can be made are the states that can be transitioned into from a parent state.

In an early experiment, this work was trained using a Markov Chain similar to that of Dahlskog et. al [45]. Trained on the rooms from *Super Mario Bros.* the state space utilized columns. The successor states are chosen with the probabilities learned from via local transition probabilities given the last known column, i.e., a true Markovian process. These single-column transitions were used for both the *Expansion* and *Rollout* steps.

The most common *Selection* strategy for MCTS is *Upper Confidence bound 1 applied to Trees* (UCT) introduced by Kocsis and Szepesvári [105]. The UCT formula for node i is:

$$UCT_i = \frac{w_i}{n_i} + c * \sqrt{\frac{\ln t}{n_i}}$$

Where

- w_i - The total score of node i - in game playing applications this is commonly the number of wins

- n_i - The number of simulations considering node i
- c - The exploration parameter. $\sqrt{2}$ theoretically, but used to tune exploration vs. exploitation in practice
- t - The total number of simulations

The main choices for a user of MC-MCTS are how the score is calculated, the exploration parameter, and the number of rollouts performed. A higher number of rollouts will better sample the space and produce a better end result, but this comes at the cost of time. In order to keep the room generation time under 30 seconds per room we limited the system to 200 rollouts per "move." We generated 320 vertical slices per room, so we could sample up to 64,000 rooms per every generated room. The exploration parameter value matters most in its relative scaling compared to the score value, so we set it to the theoretically correct $\sqrt{2}$ and focused our attentions on the score.

The score function that we used was:

$score = S + g + e + r$

where

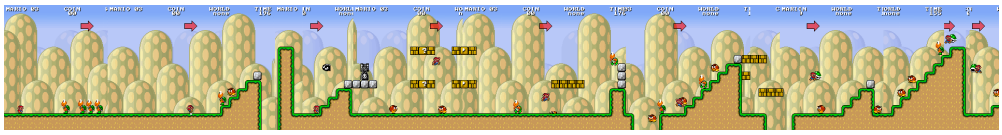
- S - Whether the room is completable or not. If it is completable then it was set to 0, otherwise it was set to a large negative number, $-100,000$ in this study.
- $g(L) = -(d_g - (16 - 2difficulty))^2$ - The desirability of the room as a function of the number of gaps. d_g is the average distance of a column to a gap. The function

is maximized when the average distance is 16 (on easy), 12 (on medium), and 10 (on hard).

- $e(L) = -(d_e - (20 - 2difficulty))^2$ - The desirability of the room as a function of the number of enemies. d_e is the average distance of a column to the nearest enemy. The function is maximized when the average distance is 20 (on easy), 16 (on medium), and 14 (on hard).

- $r(L) = -(2d_r - d_e)^2$ - The desirability of the room as a function of the number of rewards (coins or power-ups). Where d_r is the average distance is a reward. The function is maximized when $2d_e = d_r$, i.e. the distance to an enemy is twice that of the distance to a reward, biasing the generator towards rewards instead of enemies but with a semblance of balance.

(a) An easy room with no gaps and a low density of enemies.



(b) A hard room with a high density of enemies.

Figure 6.8: Screenshots of two generated rooms being played in *Infinite Mario*

In theory, these functions could be anything or could be learned via player testing. For the purposes of this work the desirability functions were quadratics that captured our hypotheses about how desired difficulty of the room would affect the desirability of the components. In general as difficulty increased so did the desirability

of gaps and enemies, but due to the quadratic nature there was a point at which too many enemies and gaps would be undesirable to anyone. The opposite was true for the rewards, as they would decrease as difficulty increased. This score was then passed through a logistic function to produce a result in the range of 0-1.

$$w_d = \frac{1}{1+e^{-score_d}}$$

Solvability was determined at a low level of fidelity for the purposes of speeding the MCTS search. Snodgrass used Robin Baumgarten's A^* controller [170] to determine whether a room can be completed or not, but it runs a full simulation of the Mario physics making it too slow for a sampling process that wants to be run thousands of times in a short period of time. We created a simplistic A* Mario controller that operates on the tile level with discretized physics extracted in a preprocessing step from the actual Mario simulation. This simulation captures running and jumping physics at fine enough fidelity to ensure that all jumps that the controller determines completable will in fact be completable in the full-fledged simulation. However, the low level controller does not consider enemies at all, so it is possible for a section of enemies to be too dense for playability. In practice, this is of not much concern as the learned probabilities make such a section exceedingly unlikely and the score function makes this unlikely as well. A sample of three rooms generated by our system can be seen in figure 6.8. Room 6.8a has a low difficulty, leading to a room with no gaps and a relative paucity of enemies. Room 6.8b is a harder difficulty which results in a higher quantity of enemies and the presence of gaps.

While the MCTS generation process makes no guarantees about global com-

pletability, it guides the sampling process in a way so as to nearly guarantee solvability, for all practical purposes. For a room to be generated that is not completable, there must be n unsuccessul "moves" chosen by the MCTS, where n is a content dependent parameter representing the smallest number of generation moves that can place a room into an unplayable state. This is an absolute lower bound, so it is not representative of any given state, but rather the worst possible state. E.g., in a tile-based *Super Mario Bros.* sampling process the shortest possible n is 6 if the current generation is the bottom tile of a row, and then a 5 tall tower is placed in the next column – making it impossible to jump over). If the sampling process produces playable rooms with probability $Pr(C)$ and the MCTS makes k rollouts per generation step, the global lower bound for the probability of producing an uncompletable room is:

$$(1 - Pr(C))^{kn}$$

i.e., the probability of producing no completable rooms for n generation steps, each of which is made up of k rollouts. For instance, the baseline 512 hidden unit, 80% dropout produces playable rooms 96.7% of the time – given a similar setup as the Markov Chain approach described, $k = 200, n = 6$ then the probability of producing an unplayable go from 3.3% to $1.6 \times 10^{-}1778$, an astronomically unlikely event. Even for a sampler that has poor guarantees on playability (e.g. the Markov Chain approach), a sampler with only 50% playability generation only produces an unplayable room with probability equal to 5.8×10^{-320}.

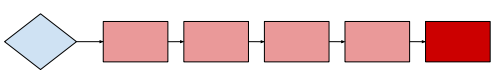
Chapter 7

Map Generation

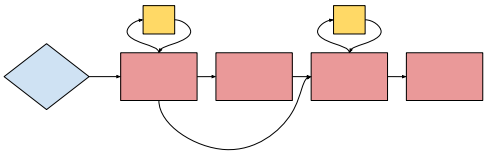
The discussion of content generation has, to this point, been focused on room generation. However, as discussed in chapter 3, the map of a game is a graph of rooms, and as such, the graph structure (the *level*) is another possible consideration for generation. Every game has *some* graphical structure underpinning it, although in some cases the graph is almost too minimal to discuss (e.g., *Asteroids* has but a single room – but a graph with one node and no edges is still a graph).*Linking Logics* are the Operational Logic governing these structures, and many other links exist in these games in ways not governing the map (e.g. there are links between the main menu or sub-menus and the rest of the game). Those links are meta-links beyond those that would be desired to be constructed.

A wide variety of graphical structures exist in games, but broadly they can be thought of in a few categories (visualized in figure 7.1) :

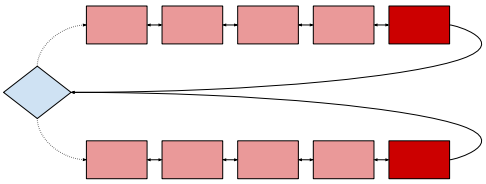
- **Linear Progression** – The player starts at the first node, and then progresses



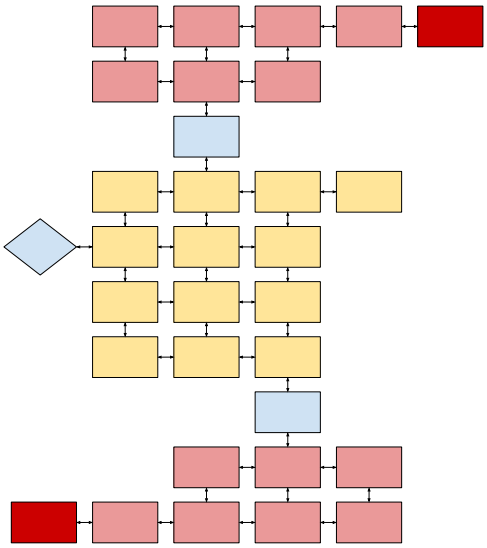
(a) The **Linear Progression** structure.



(b) The **Linear Progression with Embellishment** structure.



(c) The **Spoke and Hub** structure.



(d) The **Open World** structure.

Figure 7.1: Four common graphical structures of maps found in games. The blue diamond nodes represent the player's starting point in the game. In figure 7.1b there are digressions, small yellow squares, which lead back to the main, light red rectangles, and warps (from the first light red rectangle to the third. Figure 7.1c shows a structure going from the hub, the light blue diamond, to a "level" and then back. The dotted line represents a game where a given "level" can only be selected once (e.g. *Mega Man*). In the **Open World** example (Figure 7.1d) the "over world" is designated by the yellow nodes, and the "dungeon entrances" are the light blue rectangles.

from one room to another. This is the most common structure for early room based games (e.g., *Pac Man*, *Donkey Kong*, *Contra*, etc.). See figure 7.1a for a small illustration.

- **Linear Progression with Embellishment** – The player starts at the first node, and then progresses from one room to another. However, unlike the vanilla **Linear Progression** there are a small number of embellishments (for example, the player might exit a room go to another room and then return to the first – as in *Super Mario Bros.* when the player finds a warp-pipe to a coin room. This is the structure of many "linear" games – games commonly thought of as linear, but not strictly so (e.g., *Super Mario Bros.*, *Kid Icarus*, *Battle Toads*, etc., See figure 7.1b for a simple illustration).

- **Spoke and Hub** – A map structure that has become very common over time, the spoke and hub structure has a "hub world" to which all of the other content is connected. The content that the hub connects to is typically some other, simpler, structure, i.e., if the content were severed from the hub world, it would most likely be a **Linear Progression** or **Linear Progression With Embellishment** (e.g., *Mega Man*, *Super Mario 64*, *Spyro The Dragon*). See figure 7.1c for a simple illustration.

- **Open World** – A map structure with very high connectivity. The player has nearly full control over how they explore this structure, being able to reach a piece of content in many different ways. However, often a number of choke-points

will exist where the only way to reach one section of content must pass through a specific point (e.g. to explore a "dungeon" in *The Legend of Zelda* the player must enter the dungeon from the "over world"). In some cases, the distinction betewen **Spoke and Hub** and **Open World** can be blurry (e.g. in *Final Fantasy* pieces of land are segmented from each other during most of the game, but upon acquiring an airship the player can access any room from it, essentially converting a **Open World** structure into a **Spoke and Hub** structure). See figure 7.1d .

While all of these classes of structure exist in games, procedural generation of graphical structure has been largely focused on the **Open World** structure. For games with other structures, the generation has mostly focused on individual room structure (e.g., the generation of rooms in *Super Mario Bros.* has ignored the overall graphical structure). From a research interest perspective, this makes sense. Generally in games with less interesting graphical topology, the core variety and play space is found within the rooms. However, for games with more rich topological structures, the individual rooms have much less import, and the overall journey through the topology is where the main interest lies (e.g., in *The Legend of Zelda*, of the 236 rooms that comprise the rooms of all dungeons, there are only 99 distinct rooms – 41.9%), hence the research focus. However, this is not to say that the other structures have been ignored – Butler et al. [32] generated puzzle progressions for a fraction-teaching puzzle game, *Refraction*, i.e., the generation of a *Linear Progression* structure.

Most of the work looking at the procedural generation of **Open World** maps

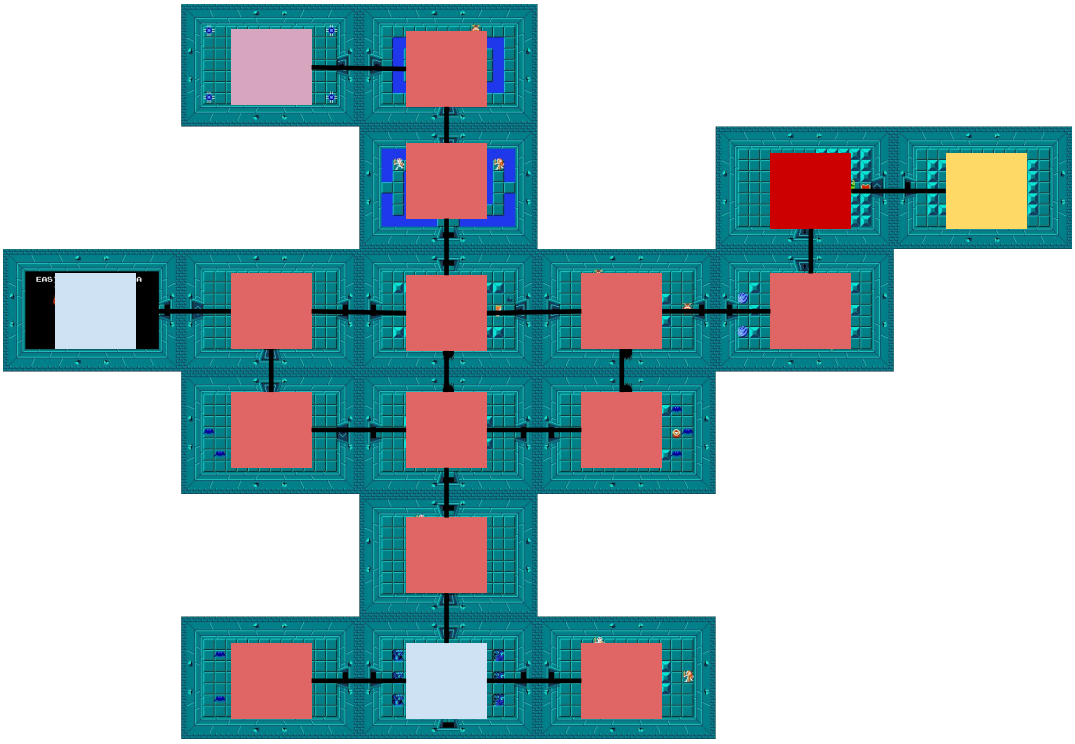


Figure 7.2: An illustration of the graphical structure underpinning a portion of the map of *The Legend of Zelda* – the "first" dungeon, aka The Eagle. The dungeon is laid out in a continuous space, but the edges between rooms – *doors* – operate in a different manner than the contiguous space inside of a room. i.e., the player can move freely within a room, but upon entering a door, they lose the ability to move freely until they have fully entered the next room.

Figure 7.3: An example of a non-Euclidean connection in *The Legend of Zelda: A Link to the Past* that defies the standard understanding of doors in the game – i.e., if a player exits from the north side of a room, they expect to enter from the south side.

has relied heavily on human authored rules and heuristics. Broadly, the existing work can be viewed as variants of searching for maps that meet the designer's beliefs about what a proper map should look like. These fall under the domains of classical search, genetic algorithms, and constraint based methods.

Classical search based methods treat the possibility space for generated artifacts as a tree and perform some search on that space to find viable maps. The system proposed by Valls-Vargas, et al. [216] performs an exhaustive breadth-first search to find all possible mission spaces for an input set of plot points. Once all possible missions have been constructed, they are evaluated for fitness based on how closely the mission matches the desired properties set forth by the designer. This approach will find the best solution, as defined by the evaluation criteria, but can only handle missions of a limited size due to the exhaustive nature of the search.

Grammar based methods are not guaranteed to find an optimal solution, but their limited expansion rules direct the search to more desirable regions of the possibility space much more efficiently than an breadth first search. Joris Dormans's system [51] uses grammars for both the mission and map spaces. The mission space is constructed by a graph grammar where nodes in the graph make up player actions such as "Defeat Boss" or "Unlock Door". From a starting seed, production rules are applied pseudo-randomly until all non-terminal nodes have been converted to terminal nodes. Once the mission space of the map has been constructed, a shape grammar is applied on the mission graph. The first node of the mission graph is taken and the system looks for a shape grammar rule that satisfies the mission node symbol, finds a location where the rule could be applied and applies it. A similar grammar based system was developed by van der Linden, et al. [112] that uses grammars for both mission and map generation as well. The advantage to van der Linden's work is that multiple mission actions can occur at the same location, e.g. kill a dragon and get dragon's treasure could both occur at the same location.

Genetic algorithm solutions have the advantage of being able to find a wide range of optimal solutions. However, they come with no guarantees that the optima found are global or that they will terminate in any amount of time. The key factors that determine the type of map created by a genetic system are the genotype and phenotype representations and the evaluation metric. The system developed by Sorenson and Pasquier [186] has direct genotype to phenotype mapping as elements in the phenotype equate to the placement of elements in the map. Their evaluation metric seeks to

keep interactions along the optimal path varied. To keep genetic diversity while also ensuring the validity of produced maps, they used a Feasible-Infeasible Two-Population split. By considering only the positioning of game elements, their system considers the mission space tangentially, by assuming that a good map space will lead to a good mission space. Valtchanov and Brown [217] created a system that uses a tree based genotype to handle room to room placement. Their fitness evaluation rewards novelty and room interconnectedness, but only considers player experience by guaranteeing the maximum optimal path length. Hartsook, et al. used a similar tree based genome, but their fitness function considers the distance between critical nodes, length of sidepaths, average number of sidepaths per non-critical node, total number of sidepaths, total number of nodes, and environmental transitional probabilities.

Horswill and Foged [87] developed a constraint solver that uses constraint propagation to place items within a map to meet a desired player experience, but it does not generate a graphical structure, instead placing items within a fixed graphical structure.

7.1 Digression on Euclidean Vs Non-Euclidean Topology

Within a room, the player moves through continuous space, but the room-to-room structure is not necessarily a Euclidean space. In some cases the graphical structure might be mostly Euclidean: e.g., in a dungeon in *The Legend of Zelda* the rooms of the dungeon are linked in Euclidean space – if the player traverses up, then

left, then down, then right, they will return to the same location (see figure 7.2 for an example). Typically, in these games, if the player leaves a room via a doorway at the edge of the room, they will enter a room at the opposite edge (e.g. leaving via the northern edge of a room and entering from the southern edge) – indicative of moving in Euclidean space – and if the player leaves the room via some other conveyance (e.g. stairs, portals, etc.), the movement is non-Euclidean; however, this is not necessarily true for all circumstances, as there are games where leaving via edges is non-Euclidean (e.g. the player might exit a room via the northern "stairs" in *The Legend of Zelda: A Link to the Past* and wind up in another room on the northern edge in a space not connected in a Euclidean manner – figure 7.3 shows this).

While many games have these non-Euclidean connections, all prior work has operated in the regime where all edges must represent a Euclidean geometry. This is more difficult, and at the least, places the constraint that all generated topologies must have a planar embedding (an NP-hard problem [209]). Furthermore, given that the individual rooms are typically of fixed size and dimension, even a planar graph might not be suitable, if a given graph can not be embedded with two connected rooms adjacent to each other.

7.2 Data Requirements

While the requirements for a room are somewhat stringent (see section 5.2), the data representation for a map is somewhat loose – in part due to the stringent

requirements on room generation. While room generation has been focused on the generation of content directly facing a player, map generation has typically operated at a level of remove, often utilizing another generation process for the individual rooms. Instead of generating the graphical structure and the room content at once, the map generation produces the structure, where each node is the input for a room generation process (e.g., generate a room that contains a key for a locked door), but the specifics of the room generation implementation (and the required input) are application specific.

For a map generator to achieve generality, the generator must be capable of generating any possible graph topology, and the downstream room generator must also be general.

For a graph generator to have generality, it must be capable of introducing new edges between arbitrary nodes. To prove this, consider a generator that only introduces new edges in conjunction with new nodes. Consider the generation move that has the highest ratio of number of edges added, e, to the number of nodes added, n. Thus, the most edges that can exist in the graph are $|E| = ke$ with a corresponding number of nodes $|N| = kn$, where k is the number of times the generation move is applied. However, this means that it is not possible for $|E| = |N|^2$ as:

$$|N|^2 = k^2 n^2$$

$$k^2 n^2 = ke$$

$$\implies kn^2 = e$$

However, since e is a fixed ratio of n, it is not possible for it to be equal to the number of times the generation move is applied, k.

However, while it is theoretically easy to allow a generator to add arbitrary edges, in practice this is beyond the scope of any generation system to date. This is due to the practical considerations inherent in the generation process, namely producing high quality content that matches the design goals. For instance, it is often the case that there will be a desired main path that the player must traverse (although there will often be content that the player can traverse if they see fit but is not essential for them to traverse) – a system that can add arbitrary edges has to ensure that the addition of an edge does not alter the graph such that this path is no longer required. In practice, all of the approaches have either limited the size of generation to make this tractable or have heavily constrained the generation space such that it is not a concern.

Chapter 8

Case Study – *Agahnim*

Given the problem of generating a map graph via machine learning there are a number of considerations for choice of approach: [1]

- The generator must consider long scale structures of the graph – e.g., is there sufficient distance between the start of the map and the end?
- The generator must consider global properties of the graph – e.g., does the map have a "good" number of nodes, edges, degree, etc.?
- The generator should be capable of supporting information about the player's traversal – e.g. backtracking is common and the degree to which it occurs is important.

The architecture used in *Agahnim* is a Bayesian Network (BN). BNs are Directed Acyclic Graphs (DAGs) representing conditional probability distributions. The

[1] Portions of this chapter originally appeared in "The Learning of Zelda: Data-Driven Learning of Level Topology"[201] and "Sampling Hyrule: Sampling Probabilistic Machine Learning for Level Generation" [196]

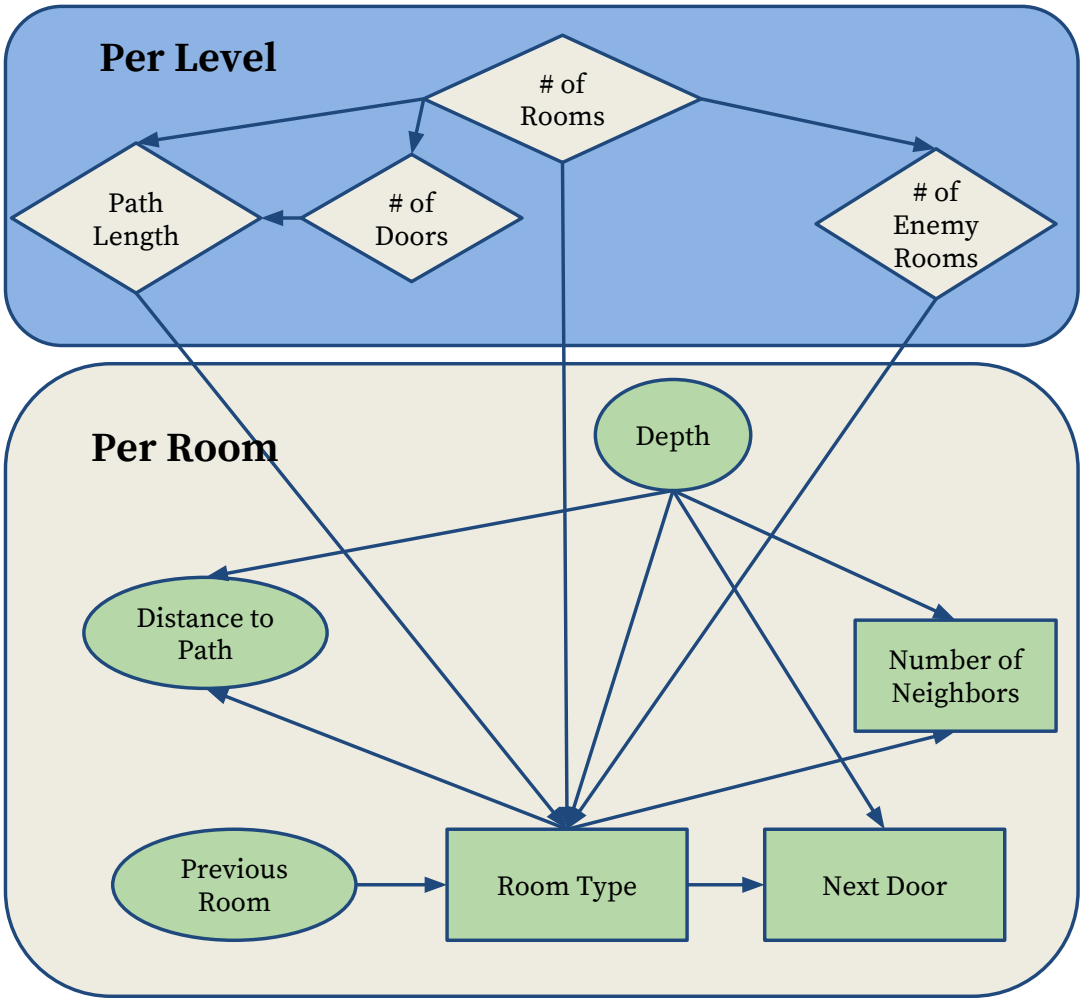


Figure 8.1: The graphical encoding of conditional probability in *Agahnim's* BN. The diamond nodes are either set by a user or sampled in order of dependency and then fixed for the duration of generation. The circular nodes are deterministically set based on the generation so far. The rectangular nodes are those that are sampled at each step of the generation.

probabilistic nature allows for easy sampling of unobserved variables – the content that is to be produced – while incorporating observed variables – high level information and deterministic data (such as player traversal information). The nature of a BN allows it to freely incorporate as many observed variables as a designer might wish, allowing for high level control of "knobs" – things such as the size of the map, the amount of backtracking, the relative frequencies of different types of rooms, etc. The BN used in *Agahnim* can be seen in figure 8.1.

	Child Distribution	
Parent Distribution	Categorical	Numerical
No Parent	$\text{Dir}(\alpha)$	$\mathcal{N}(\mu, \sigma^2)$
Categorical	$\text{Dir}_i(\alpha_i)$	$\mathcal{N}_i(\mu_i, \sigma_i^2)$
Continuous	$\sigma(x, \mathcal{N}_i(\mu_i, \sigma_i^2))$	$x \cdot \mathcal{N}_i(\mu_i, \sigma_i^2)$
Mixed	$\sigma(x_j, \mathcal{N}_i(\mu_i, \sigma_i^2))$	$x_j \cdot \mathcal{N}_i(\mu_i, \sigma_i^2)$

Table 8.1: Modeled Distributions for Parent-Child Combinations

where

- $\text{Dir}(\alpha)$ – is the Dirichlet distribution with concentration α
- $\mathcal{N}(\mu, \sigma^2)$ – is the Normal distribution with mean μ and variance σ^2
- $\sigma(a, b)$ is the Softmax function, such that $\sigma(a, b)_j = \frac{e^{a_j \cdot b_j}}{\sum_{k=1}^{K} e^{a_k \cdot b_k}}$, i.e., the probability that the child variable has category j

For child variables with numerical parents, the child is a regression – either a Softmax regression (for a categorical child) or a Bayesian linear regression (for a continuous). For child variables with categorical parents, the child variable is always a lookup table, but that lookup table might contain Dirichlet distributions (child distribution is cat-

egorical), Normal distributions (child distribution is continuous), Softmax regressions (child distribution is categorical and also has continuous parents), or a Bayesian linear regression (child distribution is continuous and also has continuous parents).

Agahnim handles both continuous, $x \in \mathbb{R}$, and categorical, $x \in \mathbb{N}$, data. The exact handling of a variable depends on the type of a random variable and the "parent" variables – the variables that the dependent variable is conditioned on. The exact handling is shown in table 8.1.

The variables considered by *Agahnim* are:

- *# of Rooms* $\in \mathbb{R}$
- *# of Doors* $\in \mathbb{R}$
- *Path Length* $\in \mathbb{R}$
- *# of Enemy Rooms* $\in \mathbb{R}$
- *Room Depth* $\in \mathbb{R}$
- *Number of Neighbors* $\in \mathbb{R}$
- *Distance to Path* $\in \mathbb{R}$
- *Previous Room Type* $\in$ { *Start*,*Enemy*,*Puzzle*,*Item*,*Boss*,*Boss Key*,*Important Item*,*End*}
- *Room Type* $\in$ {*Start*,*Enemy*,*Puzzle*,*Item*,*Boss*,*Boss Key*,*Important Item*,*End*}
- *Next Door Type* $\in$
  {*Regular*, *Bombable*, *Key Locked*, *Puzzle Locked*, *Item Locked*, *Boss Key Locked*}

Note, due to the nature of the handling of numerical values, all variables that are numerical are treated as real valued numbers. Of course, all of the considered variables are best represented as natural numbers (i.e., one can not have an irrational number of doors or rooms in a map); however, in practice, rounding to the nearest natural number has no ill effect on the generation process of *Agahnim*.

The BN underlying *Agahnim* is constructed and trained using the Infer.NET probabilistic programming framework from Microsoft Research [126] which uses the

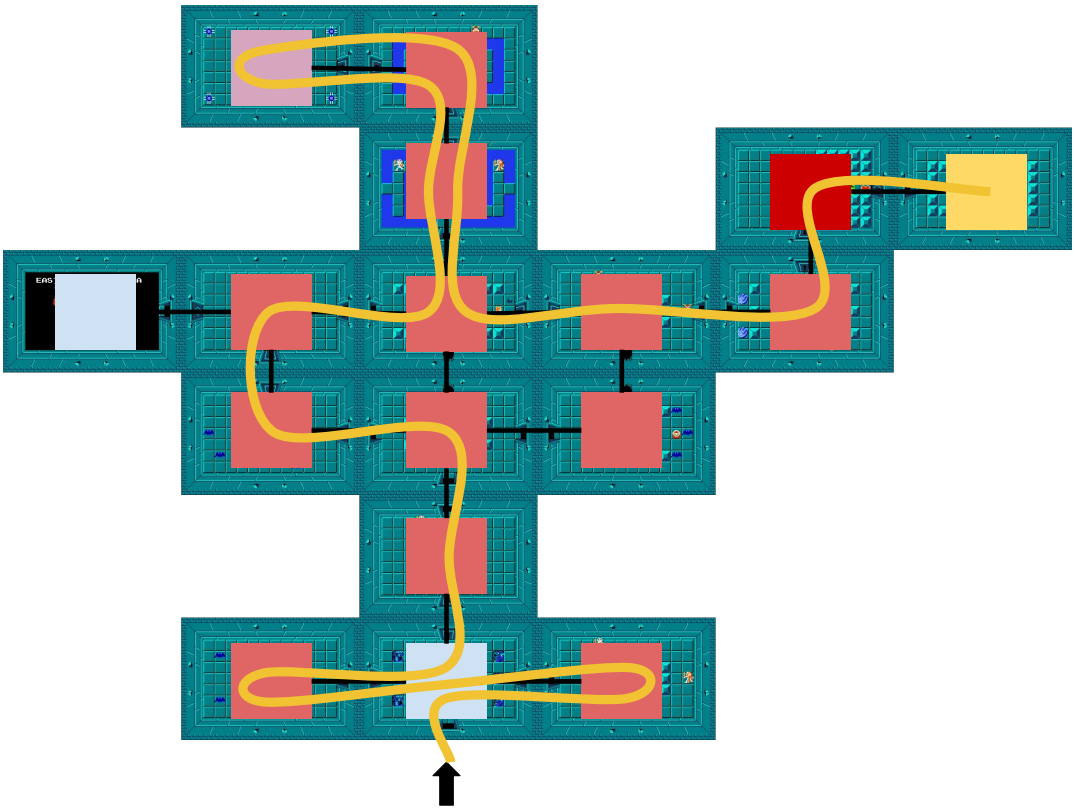


Figure 8.2: An illustration of the backtracking required in *The Legend of Zelda*. The gold line represents the optimal path (without using bombs) required for the player to find all of the important items and complete the map – which requires the player to backtrack through some rooms up to three times. If the player diverges from this optimal path, then the backtracking increases even more.

Variational Message Passing (VMP) protocol to infer the underlying distributions given a graphical model and observed data. To train, the network is constructed, the training data is "observed," and then the posterior distributions are inferred. All variables in the network are set up with uninformative priors – Uniform concentration for Dirichlet variables, $\mathcal{N}(0, 100)$ (0 mean, 100 variance Normal distribution) for the mean of the Normal distributions, and $\Gamma(1, 1)$ (Gamma distribution with $\alpha, \beta = 1$) for the variance of the Normal distributions.

While the incorporation of player traversal is critical for **Open World** map structures, it is beyond the scope of any pure-machine learned approach to date. While a generator can operate under the faulty assumption that its internal player model is correct, it is certainly not guaranteed to have a correct model (e.g., even with the incorporation of player paths, *Kamek* is still only able to produce playable paths 96.7% of the time). And unlike a game such as *Super Mario Bros.* there are no heuristics for a general graph traversal game – i.e., unlike in *Super Mario Bros.* where the generator can reasonably predict that the next player path will be to the right of the current position this is not true for a game like *The Legend of Zelda* with its high degree of backtracking (see figure 8.2 for an illustration).

To show the difficulty of this problem in a general domain, consider deep reinforcement learning. It is is capable of producing super-human game players in many games [127], but does horribly on the game included in the test set most in line with the **Open World** map structure, *Montezuma's Revenge*. Deep reinforcement learning is capable of achieving scores over 13 times higher than that of a practiced human on reaction based games (e.g., *Breakout* or *Video Pinball*), but on *Montezuma's Revenge*, where the traversal of the space is just as important as the twitch reaction, it achieves a score that is 0% of a practiced human [127]. While *Montezuma's Revenge* is a relatively complex game, the scope of its structure – 99 rooms – is well below that of a game such as *The Legend of Zelda* – 382 rooms. As general game playing progresses, this should become more tractable with time, but *Agahnim* uses a domain specific search (Dijkstra's search with a pruning of the state space such that no unnecessary backtracking occurs)

to perform its completability checks and determine the optimal path.

8.1 Generation Process

The generation process of *Agahnim* has four main phases:

1. **Observe Global Parameters** – The user either sets a high level parameter such as # of rooms, # of doors, path length, # of enemy rooms, or #total backtracks. For any variables left unset by the user, *Agahnim* samples the rest in order of dependency (i.e., # of Rooms has no predecessors, so it would be sampled first).

2. **Observe Deterministic Variables** – The deterministic variables are then set, based on the current context. The initial context has a depth of 0 and a distance to path of 0, and a previous room of *Start* (a meta room type).

3. **Sample** – The room type is then sampled and fixed. The number of neighbors is then sampled, and then for each neighbor (minus the neighbor that this room came from) the type of the door to that room is sampled and this room is then used as the context and the process goes back to step **(2)**.

4. **Resolve Inconsistencies** – After sampling until the desired number of rooms is met, *Agahnim* resolves any inconsistencies in the map, namely, if there is a locked door with no key reachable by the player. This is resolved by finding a room that is reachable by the player that is most likely to have a key (as determined by the conditional probability distribution of the BN) and setting it to have a key. This

process is repeated until such a point as there are no unresolved inconsistencies.

The purely machine learned portion comes from determining the conditional probability distribution for the sampled variables – whichever global parameters that remain unobserved by the user and the per-room variables that make up the bulk of the generation process. Some of the deterministic data is trivial to determine (e.g., the depth of a room is just one plus its predecessors depth), while some relies on the bespoke domain-specific *Legend of Zelda* solver.

8.2 Results

Given the nature of the BN as a conditional probability distribution – it can also be used to infer information about a map in a predictive manner. The BN used in *Agahnim* was compared with two other models, a more complex model that incorporated a host of other information:

- Number of puzzle rooms in the map
- Number of item rooms in the map
- Number of locked doors in the map
- Number of special locked doors in the map
- Number of doors locked by puzzles in the map
- Number of doors with soft-locks in the map
- Number of enemy rooms on optimal path
- Number of puzzle rooms on optimal path
- Number of doors on optimal path
- Number of locked doors on optimal path
- Number of special locked doors on optimal path
- Number of doors locked by puzzles on optimal path
- Number of doors with soft-locks on optimal path
- Maximum number of times a room is traversed during the optimal path
- Length from start to dungeon key
- Length from start to special item
- Neighbor types
- Door types from neighbors

As well as a much simpler model, a Naïve Bayes model, designed specifically for this experiment – predicting the number of rooms in map, given all other information. Given Bayes rule, if one wishes to determine the probability of r given the variables, this can be factorized as:

$$p(r \mid v) = \frac{p(r)\ p(v|r)}{p(v)}$$

So, Naïve Bayes is a simple model BN where the predicted variable, in this case the number of rooms in a map r, is used as the conditioning variable for all other variables, v. Since $p(v)$, the probability of the observed independent data, does not change – this can be safely dropped, and the problem becomes finding the value of r that maximizes $p(r)\ p(v \mid r)$.

Learning Method	Training MSE	MSE	MSE – no outliers
Naïve Bayes	7.74	12.06	8.65
Complex	2.89	6.40	3.41
Agahnim	2.74	6.24	3.42

Table 8.2: MSE of Inferred Room Count

The Mean Square Error (MSE) for predicting the number of rooms via the three models on a held out test set of four maps can be seen in table 8.2. The baseline Naïve Bayes approach does worse than the models that attempt to capture the dependencies – both the *Complex* model and the model used in *Agahnim* have roughly similar performance, with *Agahnim* having better training and test performance, although performing worse when an outlier map is removed from the test set. The outlier is the largest map from *The Legend of Zelda: A Link to the Past*, being nearly twice the size of any map in the training set, i.e., not a representative map.

Learning Method	Negative Log-Likelihood
Naïve Bayes	3655.5
Complex	187.8
Agahnim	17.9

Table 8.3: Perplexity for the three models.

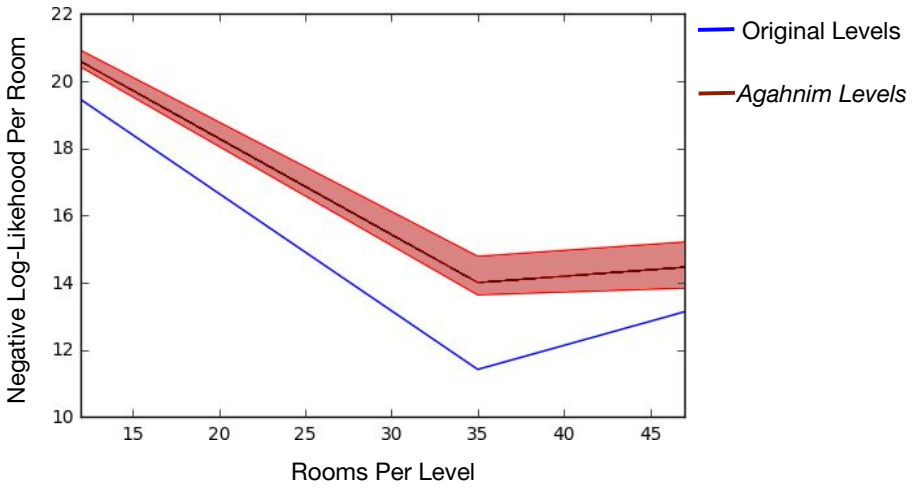
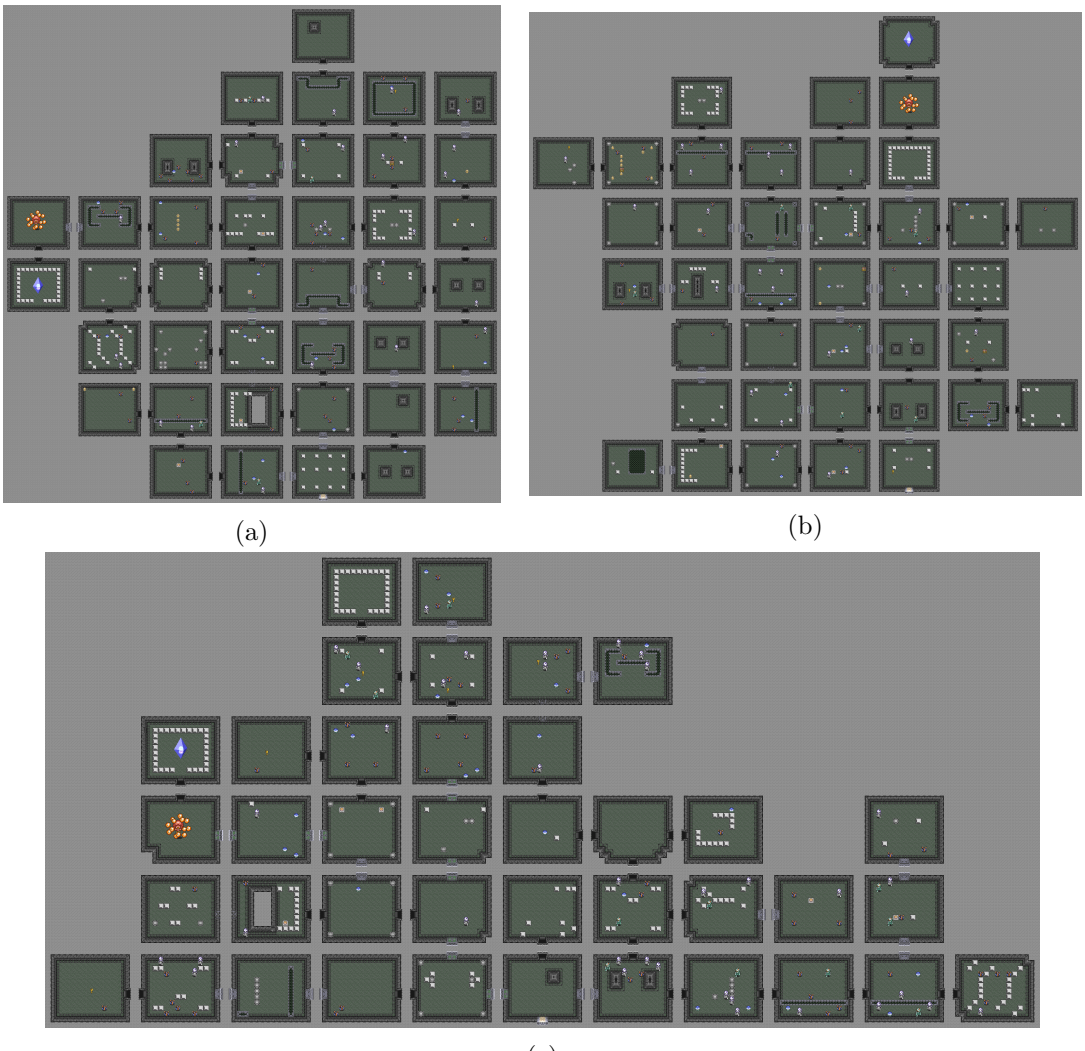


Figure 8.3: Negative log-likelihood per room versus rooms per map for both the generated maps and all of the original maps (training and test set). The dark red line is the median for *Agahnim*, and the spread represents the 25^{th} and 75^{th} percentiles.

The overall negative log-likelihood of the models on the held out set can be seen in table 8.3 (smaller negative log-likelihood represents a more likely model given the data). *Agahnim* out-performs the others by at least an order of magnitude indicating that, of the considered distributions, it is the best at the predictive task for all considered variables.

Unlike *Kamek*, which is able to produce rooms quickly ($\approx$ 6 seconds per room on a GTX 980M), *Agahnim* is a slow generation process, taking $\approx$ 1.5 minutes per room which means that the generation of a full map takes between 15 minutes and an hour. As such, the results are of much smaller scope than *Kamek*, with 20 maps



(a) (b)

(c)

Figure 8.4: Representative sample of three generated maps from *Agahnim*.

generated for each of three different settings for number of generated rooms: 12 (the minimal found in the training set), 35 (the median found in the training set), and 47 (the maximal found in the training set). The negative log-likelihood per room can be seen in figure 8.3. While, as discussed in section 6.1.1, maximizing the likelihood of generated results is not a guarantee for high quality results, it is a good proxy, with an increase in likelihood generally being correlated with an increase in quality (as shown in table 6.4). The negative log-likelihood per room versus the number of rooms per map for both the generated maps and all of the original maps (training and test set) is shown. In general, maps around the median size are the most likely, which is expected, as that size is generally more likely than the other sizes. The human generated maps are more likely than the generated, but not to an unreasonable degree (e.g. the 25^{th} percentile of likelihood for the maps with 35 rooms are more likely than the human produced map of size 25). A sampling of maps generated by *Agahnim* can be see in figure 8.4.

8.3 Discussion

Agahnim represents an early experiment with map generation, and, as such, a number of refinements are possible. The largest issue with *Agahnim* is its lack of generality. *Agahnim* generates maps room-to-room connections, but is never able to produce edges between rooms more than one step away in the generation process. This leads to *Agahnim* producing a tree-like maze with no cycles, a structure that is rather

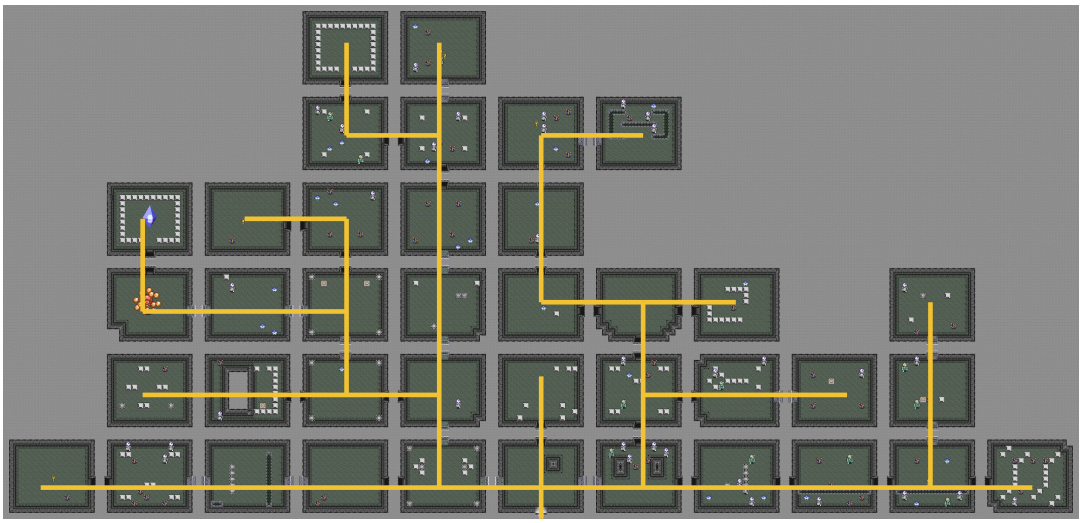


Figure 8.5: A highlighting of the connectivity for a map generated by *Agahnim*. The lack of cycles is a problem with the generality of *Agahnim*.

unlike that of the input set of maps from *The Legend of Zelda Series* – an illustration highlighting this lack of cycles can be seen in figure 8.5.

Further harming the generality of *Agahnim* is a lack of orthogonality in its representation of room type. The annotation scheme used in the training set for *Agahnim* is orthogonal in its covering of room type; i.e., a room is a set of binary flags indicating whether a room has a given type (e.g., a room type might be $\{Enemy\}$, $\{Enemy, Item\}$, $\{Item, Puzzle\}$, etc.). However, *Agahnim* collapses these orthogonal Binomial variables into one Categorical variable (a la the discussion of entity annotations in section 3). E.g., while a room can have the types *Important Item* or *Key*, the combined category of *Important Item-Key* does not exist in the original dataset, so it is impossible for *Agahnim* to generate a room with that type (although *Important Item-Key-Puzzle-Enemy* does exist in the original dataset).

On the practical side, the slow generation process hinders large-scale evaluation of the generative space of *Agahnim*. Unlike *Kamek*, producing a large volume of maps is beyond the practical scope of what is feasible (e.g., producing 1000 maps would take *Agahnim* $\approx$ 36.5 days). As such, the evaluation of *Agahnim* will not be considered in further detail in Chapters 11 and 12.

Part III

The Evaluation of Generative Methods

Chapter 9

Existing Methods for Generative Evaluation

While the foremost goal of any procedural generation system is that of producing high quality, novel content there is still no agreed upon methodology for assessing whether a generator has achieved that goal. The computational creativity [5] community has focused on not just the content generated, but also the underlying processes of the generators. These processes were discussed somewhat in section 5.2, but are not the subject of the following chapters. What follows is an assessment of the existing methodologies researchers use to understand the capabilities of a generator, by visualizing the generative space, by comparing and contrasting generators, and by presenting pieces of content. While these practices are useful stepping stones, there are gaps in what they present and what questions they can fundamentally answer. In the following chapters I will discuss new methodologies for assessing the capabilities of generators –

with a specific eye towards PCGML generators – both in the large scale assessment of the generative space (Chapter 11) and in small scale selection of individual pieces of content (Chapter 12.1) .

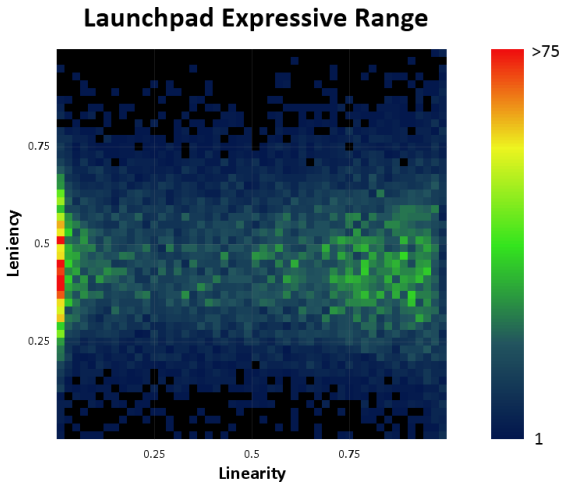


Figure 9.1: An example expressive range plot from Smith's dissertation [179]. Brighter regions represent a higher probability density for the generator's sampling distribution.

In the domain of procedural generation of game level content, expressive range has been the dominant qualitative exploration tool. Expressive range is an idea put forth by Smith and Whitehead [177, 178] whereby two metrics are chosen (e.g., for platformers Smith and Whitehead settled on *linearity*, a measure of how closely the ground of a level follows a line, and *leniency*, a proxy for how difficult a level is) as the axes for a density plot, showing which areas of the generative space a generator tends to explore. While the metrics that make up the expressive range are usually quantitative in nature, the act of the analysis is qualitative, in that it relies on the subjective analysis of the reader.

Danesh, a tool from Cook et al. [42], is designed to help bridge the gap

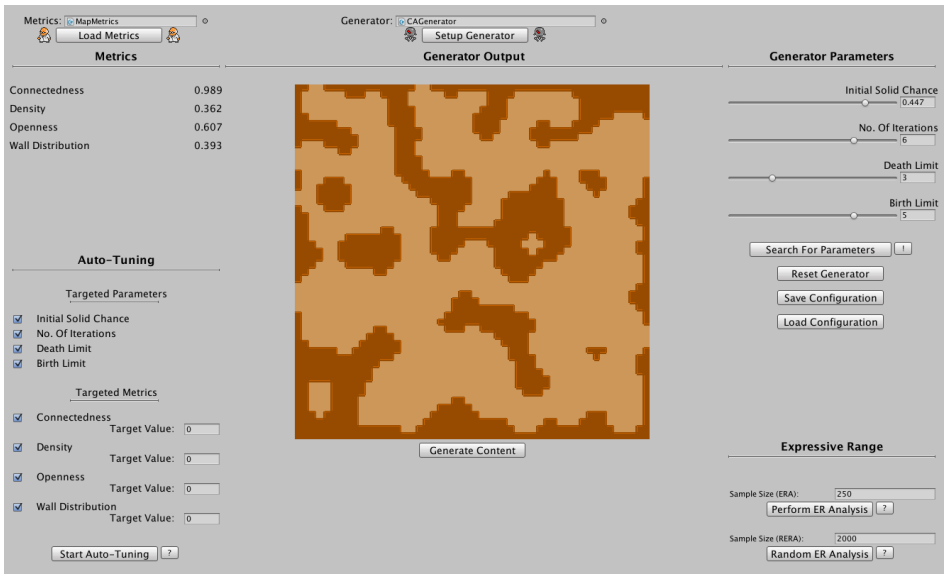


Figure 9.2: *Danesh* in situ. *Danesh* is a plugin for the Unity engine. Reproduced from [42]. Pictured is a map generated by an algorithm, the parameters for the generator (top right), the metrics of the map (top left), tools for targeting metrics (bottom left), and tools for generating expressive range analyses (bottom right).

between a generator and its expressive range. For many generators, it can be difficult to determine which set of parameters will achieve the desired outcome. *Danesh* displays the expressive range of a generator and allows users to target areas of the expressive range they find desirable, and then tries to find parameterizations that target that region of the expressive range. Figure 9.2 shows an example of Danesh in action.

In Smith's thesis [179] she discusses the expressive range of *Launchpad* [178], stating "There is good coverage of the generative space for both linearity and leniency." However, this claim goes unexplored. Why is the coverage (as shown in figure 9.1) "good?" This question is tangentially addressed by Teng [208]. Exploring the expressive range of his generator, he finds a region of the expressive range of two metrics (the connectivity and density of road networks) goes uncovered by his generator (e.g., the

generator can not produce road networks of high density and low connectivity, as these properties are entangled); however, this is not a failure of the generator, but rather a sign that the generator is producing plausible road networks – a victory for the approach. As such, it is incorrect to say that a large expressive range is unequivocally a good quality for a generator.

Of course, this perhaps feels at odds with the earlier discussions of generality. If it is an unequivocal good that a generator have generality, why is it not an unequivocal good for a generator to have a large expressive range? Remember, it is an unequivocal good for a generator to be able to generate all possible content (i.e., be capable of supporting a large expressive range), not to actually have a large expressive range. Generality is important only for preferring a more general generator to another *assuming they produce content of equivalent quality.* One major aspect of the quality of a generators 'content – for machine learned approaches, certainly – is matching the distributions of the training data.

The goal of a machine learned approach is for the generator to learn the style latent within the training data, so as to produce content that could conceivably pass the test of "Was this content plausibly in the training data?" As such, it is important to understand the expressive range of not just the generator, but of the training corpus. If the expressive range of a generator differs from that of the training set, then it speaks to a fault of the generator that is "flavoring" the generated content.

A practice that is common in the field of procedural content generation is to pick some subset of metrics (since most research is in the platformer domain, most pick

the *linearity* and *leniency* metrics put forth by Smith and Whitehead [177]), and, instead of displaying the expressive range density plots, list summary statistics for the generator [45, 129, 165, 189, 118, 117]. However, while the pragmatics are understandable, as authors have limited space to report results and large scale comparisons require a table to report all combinations; however, for generator evaluation, this is an unsatisfactory approach. The shape and density of the sampling distributions is important, and simple measures such as mean and standard deviation (and by extension, tests such as Student's t-test [191] which rely on said measures) tell a very incomplete story.

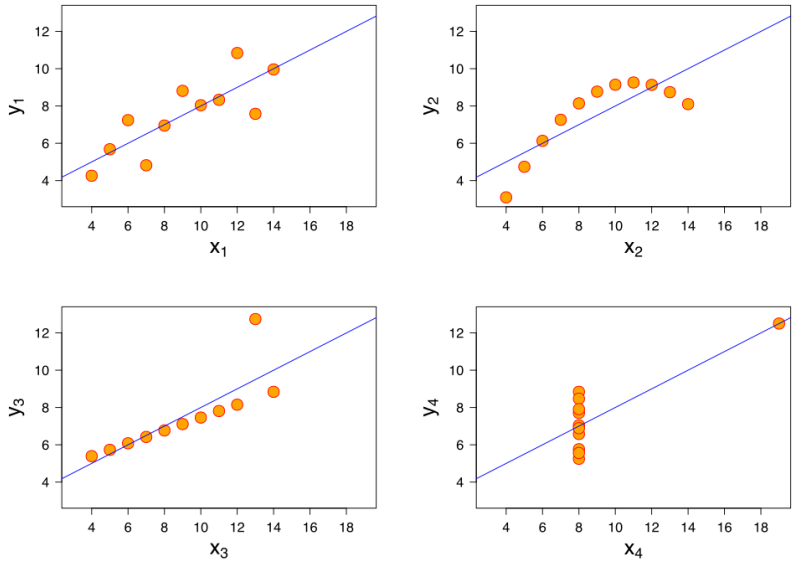


Figure 9.3: Anscombe's Quartet showing four distributions with very different visual qualities but (nearly) identical summary statistics. Reproduced from [3].

Anscombe's Quartet [21] is quartet of datasets, each demonstrating pairs of $< x, y >$ coordinates that all have the same mean, variance, correlation coefficient, maximum likelihood linear regression, and coefficient of determination for the linear

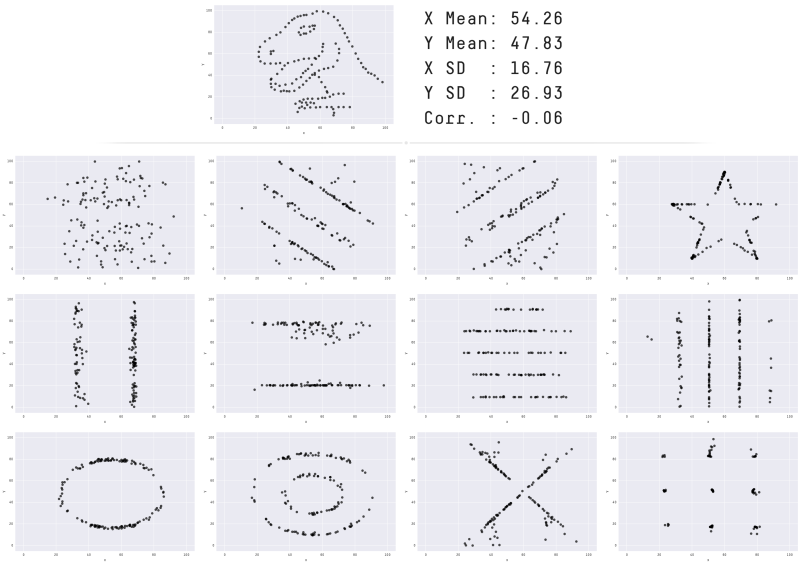


Figure 9.4: The Datasuarus Dozen [120] with nearly identical summary statistics. Reproduced from [120].

regression. However, upon visual inspection, it is quite easy to see that the the data-generating distribution for the four examples is quite different (as shown in figure 9.3). This has been taken further with work by Matejka and Fitzmaurice [120] who produced a method whereby a distribution of data can be perturbed to match another image, while keeping the same summary statistics. The so-called Datausaurs Dozen (as they were all generated from the source dinosaur distribution) can be seen in figure 9.4. This speaks to the danger of only reporting statistical properties, showcasing the need for the visual inspection of the sampling distributions (discussed in section 11.1) and qualitative evaluation of both generated artifacts (discussed in section 12.1). However, while summary statistics are not sufficient, it is desirable to have statistics that allow for the comparison of generators' distributions, but these statistics must be holistic, so as to actually capture the shape and density of the distributions and can not just be

univariate measures (discussed in section 11.3).

As generative models have become more popular in the image generation domain, the desire to find metrics that determine whether a generator is achieving its goal have begun to be researched by that community. One such measure that has become the de facto standard for Generative Adversarial Network based approaches is that of Inception Score (IS) [158], which has seen widespread adoption [99, 226, 88]. IS utilizes a pre-existing image classification model, the Inception v3 Network [204] pre-trained on the ImageNet corpus [48], which has been trained to determine the class of the object found in the image (e.g., Does the image contain a dog, a cat, a flower, a car, an airplane, etc.?). Given this model, and its ability to produce a probability distribution over the categories, $p(y)$, the IS of a generative model, G, is defined as:

$$\text{IS}(G) = \exp(\mathbb{E}_{x \sim G} D_{KL}(p(y|x)||p(y)))$$

Where $x \sim G$ means that image x was sampled from generator G and $D_{KL}(p(y|x)||p(y))$ is the Kullback-Leibler divergence between $p(y|x)$ and $p(y)$. Kullback-Leibler divergence [107] is measure of the relative entropy between two distributions, i.e., how much they diverge from each other. Intuitively, the IS tries to balance two measures, $p(y|x)$ and $p(y)$. It tries to achieve high $p(y|x)$, i.e., it wants to be able to produce images that are capable of being easily classified by the supplementary classification model. It also tries to achieve high entropy in $p(y)$, i.e., produces a wide range of images that cover all possible categories.

However, while the approach is appealing (and has been shown to correlate well with human perceptual ratings [158]), it has two limitations that make it unappealing for

procedural game content generators. First, it relies on a pre-existing vetted classification system. While image classification has been an area of research for decades, there is no such community (or even an analogue for a similar problem domain) in the field of games. Any such classifier would, almost by necessity, have to be trained in a bespoke manner for every new game, and would lack the external validation that a state-of-the-art network like the Inception v3 comes ready with. However, while this practical consideration likely precludes using a measure such as the Inception Score (or similar Inception based measures such as the Fréchet Inception Distance [80]), there are fundamental flaws in the Inception Score as discussed by Baratt and Sharma [25], namely that it is easy to find pathological cases where horrible generation distributions produce a higher IS than the true underlying distributions – meaning that the score can be easily gamed and does not measure how well a generator is matching the true generative distribution, the ultimate goal for a machine learned generator.

In a rambling (her words, not mine) [176], Smith discusses pitfalls (i.e., deadly sins) that procedural content generation research falls into. The cardinal sin is that of cherry-picking a handful of generated artifacts and showing them in lieu of actual results. This approach is problematic in its selection criterion. Why is this content being shown? Is it actually representative? Is it the best result that the generator produced? This is especially important to determine for generators where *mode-collapse* is a concern. Mode collapse refers to the phenomenon where a generative system (typically a Generative Adversarial Network) repeatedly samples similar portions of the generative space, even with different parameterization and random seeding. This is obviously an undesirable

property for a generator, as one wants different random seeding to produce varied results, but this is an insidious problem that can go undetected by the cherry-picked display of results. A shown image (or set of 5, 10, etc.) will not display the problem of mode collapse and any random selection of n images might still not display the problem. When a generator is supposed to be able to generate an unbounded (or, as is common, 4,294,967,295 possible outcomes – the size of an unsigned 32-bit integer) even showing something like 500 or 1,000 pieces of generated content is a paltry sampling (showing only $0.00001 - 0.00002\%$ of the supposed generative space). It is critical to have a principled way of showcasing the generative content, such that cherry-picking for the best results or to avoid showing mode-collapse is not possible. This is discussed in more depth in section 12.1.

Chapter 10

Aside – A Discussion of Metrics for Platformers

Before discussing methodologies based on evaluation metrics for PCG systems (Chapters 11 and 12), it is important to gain an understanding of the metrics themselves. The generation of platformer rooms has been the largest area of research within the domain of PCG, and the amount of work on computational metrics to evaluate platform game rooms [177, 34] significantly lags behind the amount of work concerning room generation [175, 181, 198, 45, 166, 75, 160, 152, 117], to name a few examples of PCG systems for generating rooms of platform games. Furthermore, while a large number of metrics have been proposed and a smaller number actually used, what these metrics *actually mean* is mostly unexplored.[1]

To try to find out how metrics actually correlate with human understandings

[1]Portions of this chapter originally appeared in "Understanding Mario: An Evaluation of Design Metrics for Platformers"[195]

of platformer rooms, I use the dataset described by Reis et al. [152].[2] Reis et al. used the Notch Level Generator (NLG),[3] to generate a library of 2,000 rooms of size 20×15 (a typical *Super Mario Bros.* room is approximately 10 times longer than Reis et al.'s small rooms).

NLG receives as input a difficulty value d for stochastically determining the number of enemies to be included in the room. Reis et al. used NLG to generate rooms with different values of d to ensure diversity in the dataset produced. These rooms were made available online for evaluation, and volunteers played 1,437 distinct small rooms and then provided 2,715 evaluations. The small rooms were evaluated according to the volunteers' perceived visual aesthetics, enjoyment, and difficulty on a 7-point Likert-like scale. The evaluations were obtained in 125 different sessions of play. A session of play is defined by a volunteer entering the system, annotating a collection of small rooms, and exiting the system. Since Reis et al. wanted to maximize the number of annotated small rooms, in order to simplify the annotation process, they did not ask for the volunteer's identity nor their demographic information. The number of sessions of play offers a reasonable approximation of the number of volunteers who participated in their data collection.

Two independent volunteers agreed to contribute non-anonymously to Reis et al.'s data collection. The ratings provided by these two volunteers allow us to perform an inter-rater study (the two volunteers evaluated 453 rooms in common). This subset of 453 evaluated rooms is used to verify how well a volunteer is able predict the ratings

[2] Available at `http://www.dpi.ufv.br/~lelis/downloads/Mario-Dataset.zip`

[3] The system is named after Markus "Notch" Persson.

of another independent volunteer. Also, one of these two volunteers evaluated 38 rooms twice. These ratings are used to evaluate how the evaluations of a single person correlate with this person's own evaluations.

Applying metrics to generated rooms has been a common practice since Smith and Whitehead [177] introduced linearity and leniency. Horn et al. [86] extended these metrics with density [165] and pattern density [44] (the number of times certain meso-patterns appear in the room). Canossa and Smith [34] extended these metrics with a proposal for many more that attempt to address the complexities of properties of interest, such as aesthetics and difficulty.

There has been less work on mapping these metrics back to actual human affective responses. Pedersen et al. [145] predicted human responses using mostly features of the players' playtraces in addition to metrics related to the gaps in the rooms (number of, width of, etc.). Summerville et al. [202] used playtrace metrics in addition to metrics related to the frequency of gaps, enemies, and rewards to predict players' responses. The most similar work is that of Mariño et al. [118] who used some of the metrics used by Horn et al. [86] to predict the perceived difficulty, enjoyment, and visual aesthetics of generated rooms.

10.1 Previous Computational Metrics

As mentioned, a number of metrics have been previously used, and as such it is important to see how these metrics actually map back to human affective response.

Following are a number of metrics that have been previously defined, which will be utilized in the analysis in section 10.3.

Linearity: The linearity of a room is computed by performing a linear regression on the center points of the platforms and mountains contained in the room [177]. The linearity is the average distance between the center point of platforms and mountains in each column of M and the linear regression's line. The linearity values are first multiplied by -1 (so higher values indicate more linear rooms) and then normalized into the range of $[0, 1]$.

Leniency: Leniency approximates how much challenge the player experiences while playing a room [177]. The leniency of a room is the sum of the lenience value $w(o)$ of all objects o in G: $\sum_{o \in G} w(o)$, normalized by the width of M. This uses the lenience values specified in previous works [165, 118]. That is, power-up items have a weight of 1, cannons, flower tubes, and gaps of -0.5, and enemies of -1. The average gape width of the room is subtraced from the the resulting sum as defined by Shaker et al [165]. The leniency values are first multiplied by -1 (confusingly, larger leniency values indicate more challenging rooms) and then normalized into the range of $[0, 1]$.

Density: Some objects can occupy the same x-coordinate in M (e.g., mountains in SMB can be 'stacked-up" together). The density of a room is the average number of "mountains" (sections of solid terrain above ground room) occupying the same x-coordinate in M [165]. Density values are also normalized into the range of $[0, 1]$,

where values closer to one indicate denser rooms.

Negative Space: Negative Space is the percentage of the empty space that is reachable by the player [34]. Jumping in platform games such as SMB is the core way for players to navigate the vertical space. A higher Negative Space metric often means more "floating" platforms and mountains which tend to be more enjoyable and aesthetically pleasing than simply progressing along the ground.

Other Metrics: In addition to negative space, Canossa and Smith [34] introduced 19 other metrics, which are not used in this study. Their metrics are categorized into: *aesthetic*, *difficulty*, *topology*, and *strategic* metrics. Aesthetic metrics cover aspects such as music and the visual palette. Difficulty metrics expand on leniency by categorizing the room's source of difficulty. Topology metrics look at the physical space of the room and measure relevant features. Strategy metrics focus on how a player will/must react in a room.

10.2 Novel Computational Metrics

In addition to the previously defined metrics, a number of other metrics (including some base-line metrics) are considered for evaluating game rooms.

Symmetry (S) The notion of symmetry has been empirically shown to correlate with the visual aesthetics of graphical user interfaces [133] and images [26]. The model of symmetry is based on the work of Ngo et al. [134] and Mariño and Lelis [117].

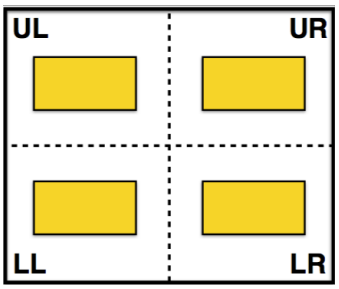


Figure 10.1: Example of a symmetrical image.

The symmetry of a room is computed by dividing M into four equal regions by a vertical and a horizontal separation line. The resulting regions are named Upper Left (UL), Upper Right (UR), Lower Left (LL), and Lower Right (LR). Let $X(LL)$ be the sum of the distances between the center of all objects in region LL and the vertical line; $Y(LL)$ be the sum of the distances between the center of all objects in LL and the horizontal line; and $A(LL)$ be the sum of the areas of all objects in LL. The symmetry value S of a room M is defined in terms of functions $X(M)$, $Y(M)$ and $A(M)$, defined below:

$$\begin{aligned} X(M) = &|X(UL) - X(UR)| + |X(LL) - X(LR)| \\ &+ |X(UL) - X(LL)| + |X(UR) - X(LR)| \\ &+ |X(UL) - X(LR)| + |X(UR) - X(LL)| \,. \end{aligned}$$

The value of $X(M)$ accounts for the "symmetrical" distance across the vertical line, across the horizontal line, and across the vertical and horizontal lines. The values of $Y(M)$ and $A(M)$ are defined analogously by using Y and A-values instead of X-values.

The S-value of a room is defined as as follows:

$$S(M) = X(M) + Y(M) + A(M)\,. \tag{10.1}$$

S captures the intuitive notion of symmetry illustrated in Figure 10.1, where the yellow rectangles represent objects in the room. The room shown in Figure 10.1 has an S-value of zero, which means that the room is perfectly symmetrical according to S. The S-value of the room is zero because there are objects with exactly the same area in each region. Also, the objects in regions UL and LL are at the same distance from the vertical separation line as the objects in regions UR and LR; and the objects in regions LL and LR are at the same distance from the horizontal separation line as the objects in regions UL and UR.

Balance (B) According to Ngo et al. [133], the metric of balance measures whether the objects the player might find interesting, and thus attract their eyes, are well distributed in M. The assumption is that the "attractiveness" of an object is proportional to the object's distance to the horizontal separation line as well as the object's area.

Balance is computed by dividing M into two regions, Top (T) and Bottom (B), of equal size. In Figure 10.1 T is defined by the union of regions UL and UR, and bottom by the union of regions LL and LR. G_T and G_B are the set of objects in T and B, respectively. The Balance value of a room is computed in terms of function W,

which is defined for the objects in region T as follows:

$$W(G_T) = \sum_{o \in G_T} dy(o)A(o)$$

where $dy(o)$ is the distance between the center of the object o and the horizontal separation line, and $A(o)$ is the area of o. $W(G_B)$ is defined analogously. The balance value of a room M is then defined as the absolute difference between $W(G_T)$ and $W(G_B)$:

$$Balance(M) = |W(G_T) - W(G_B)|$$

Reachability Reachability measures the proportion of elements placed in M that are reachable by a player, i.e., that the player can directly interact with. The hypothesis is that players rate poorly the visual aesthetics of rooms that have fundamental flaws such as unreachable objects. The reachability is calculated as follows:

$$R(M) = \frac{n_{RC}}{n}$$

where n_{RC} is the number of unreachable elements, and n is the total number of objects in M. The value of n_{RC} can be computed by applying domain-specific rules (e.g., in SMB Mario is unable to jump more than a given number of tiles).

Decoration Frequency. Rooms are composed of many different objects, and the grid M tends to be sparse, with most of the grid cells being empty. Some objects, such as

the question-mark blocks, pipes, or enemies bring more visual variety to the room (i.e., they "decorate" the room), and as such the decoration frequency metric is the number of decoration tiles divided by the size of the room:

$$DP(M) = \frac{\sum_{x=1}^{w} \sum_{y=1}^{h} \text{pretty}(M[x][y])}{w \times h}$$

where pretty(t) is defined as being equal to 1 when t is any of the following tile types: Pipe, Enemy, Destructible Block, Question Mark Block, or Bullet Bill Shooter Column and 0 otherwise.

Tile Frequencies This metric is simply defined as the number of tiles of that type divided by the size of the room,

$$EP(M) = \frac{\sum_{x=1}^{w} \sum_{y=1}^{h} \text{Type}(M[x][y])}{w \times h},$$

for each of 13 different types: Solid, Enemy, Destructible Block, Question Mark Block With Coin, Question Mark Block With Power-up, Coin, Bullet Bill Shooter Top, Bullet Bill Shooter Column, Left Pipe, Right Pipe, Top Left Pipe, Top Right Pipe, and Empty. These are included as they represent a base-line. More complex metrics tend to use these in different combinations and scalings (e.g., Leniency incorporates the number of enemies), but it is important to see if they hold more power than the simplest possible metrics.

Tile Position Summary Statistics. The distribution of object types in a room contains important information about the experience the player will encounter. For example, rooms with more variance in the height of ground tiles will likely require the player to jump more. Rooms with low variance on the x-coordinate where the enemies are placed will likely have a closely packed group of enemies. For each of the 13 tile types:

- μ_x and σ_x – The mean and standard deviation x position of that tile type.
- μ_y and σ_y- The mean and standard deviation y position of that tile type.

Enemy Sparsity (ES) This metric is a measurement of whether the enemies are grouped together or spread in the room. The enemy sparsity of room M is computed as follows.

$$ES(M) = \frac{\sum_{e \in \mathbf{E}} |x(e) - \bar{x}|}{|\mathbf{E}|}.$$

Where $\mathbf{E}$ is the set of enemies in M, $x(e)$ is the x-position of enemy e in M, $\bar{x}$ is the average x-position of all enemies in $\mathbf{E}$, and $|\mathbf{E}|$ is the total number of enemies in M.

Enemy Sparsity is similar to the metric μ_x for enemy tiles as they both measure the horizontal spread of enemies in the room. The difference between the two metrics is subtle: while the former computes the spread with the standard deviation formula, the latter uses the absolute differences between enemies and $\bar{x}$. This subtle difference is discussed further in Section 10.3.5.3.

Tile Indicator. Some of the above metrics only make sense if a given tile type is present in a room. For each tile type, this metric is defined as 1 if the tile type is present in the room and 0 if it is not.

Path Length Percentage (Path %). This metric is the proportion of the room that is taken up by a path from beginning to end (i.e., a sequence of grid cells from Mario's initial grid position to a grid position after the finish line of the room) found by an A^* search [77]. The expectation is that the more obstacles that are in the room, the longer the required path. This is because the player will need to move around the grid to avoid the obstacles. The Path % metric of M is computed by dividing the number of grid cells in the path found by A^* for M divided by the total number of tiles in M ($w \times h$).

Jump Count. Using the same A^*, this is simply the count of the number of jumps required to complete the room. If all actions a player can issue (e.g., move, jump, etc.) have cost of one, an A^* search will minimize the number of actions required to finish the room (i.e., for A^* all actions are equally costly). This metric shows only the number of jumps required to finish the room, not the number of possible jumps, which can be very large. In fact, an A^* search minimizing the number of actions could return a sequence of actions that includes jumps that are easily replaced by runs. In order to find a sequence of actions that includes jumps only when necessary, the jump action costs more than all other actions. In the A^* implementation a jump action costs 2 while all other actions cost 1. This means that the sequence of actions encountered by A^* to finish the room

will run/walk if possible, and only jump when a gap/enemy/hill requires it.

10.3 Empirical Results

The problem of predicting human ratings is a classification task which I tackle with a multinomial LASSO regression [211]. As a byproduct of its regression, LASSO also selects a subset of discriminative metrics for the multinomial regression task. Then, each metric selected by the LASSO regression is correlated with the human ratings.

I chose to use a multinomial regression instead of the more standard linear regression due to the nature of the ratings. While the ratings are Likert-like (i.e., they have a number associated with them and are not purely categorical responses such as "Poor" or "Great"), it is likely incorrect to make any assumptions about the scaling (i.e., the difference between 1 and 4 might not be the same as the difference between 4 and 7).

10.3.1 Metrics Selection with LASSO

In this first experiment I perform a 10-fold cross-validation multinomial LASSO regression for each criterion: difficulty, visual aesthetics, and enjoyment utilizing the full dataset with all raters. I chose multinomial LASSO for two reason **(1)** I believe the Likert style data should not be treated as interval (multinomial) **(2)** I wanted a regularization technique that encouraged sparsity for variable selection (LASSO as opposed to ridge or elastic net regression). In addition to linearity, leniency, density, and negative space, I use all 85 metrics described in this experiment.

A multinomial LASSO regression minimizes the categorical cross entropy while limiting the absolute sum of the regression coefficients scaled by an input parameter λ. Many of the regression coefficients are set to zero due to the λ limitation—LASSO performs feature selection during its regression procedure. A multinomial regression predicts a class, k, from a set of K classes for each data point, x, by taking the class with the highest probability.

$$\Pr(k|x) = \frac{e^{\beta_k x}}{\sum_{l=1}^{K} e^{\beta_l x}} .$$

The multinomial LASSO regression effectively predicts, for a given room M, which of the 7 "Likert classes" M belongs to. Note that due to the fact that there is disagreement among the human raters it is impossible for the regression to achieve no error (e.g., Rater A gave a room a 3 while Rater B gave it a 5 means that the regression can get at most one of those correct).

Many of the metrics introduced in this paper encode similar information. For example, the percentages for tile types Left Pipe and Right Pipe are expected to provide similar information. The goal of this experiment is to select a subset of discriminative metrics for the task of predicting each of the evaluation criteria.

One LASSO regression is performed for each of the three criteria where the λ is chosen to be the the maximal λ parameter that was within one standard error of the minimal training error; the minimal training error is achieved by including all metrics. LASSO reduced from 89 metrics to only 12 metrics for difficulty, 16 metrics for visual

aesthetics, and 14 metrics for enjoyment.

10.3.2 LASSO Prediction Results

The performance of the multinomial regressions can be seen in Table 10.1 in terms of *accuracy* and *mean absolute error* (MAE). The accuracy is computed as the percentage of rooms classified correctly (assuming the ground truth for room M is the median rated value for M), and MAE is the mean absolute difference between the predicted values and the ground truth values. The LASSO model achieves an accuracy of 37.6% in difficulty, 33.1% in visual aesthetics, and 35.2% in enjoyment. A random classifier is expected to achieve an accuracy of ≈14.3% as the problem has 7 distinct classes.

The MAE values of the multinomial regression predictions vary from 1.16 (difficulty) to 1.29 (visual aesthetics), which means that the prediction model errs on average slightly more than one point in the 7-Likert scale. The 7-Likert scale is defined by the following points: 1 (strongly disagree), 2 (mostly disagree), 3 (somewhat disagree), 4 (neither agree nor disagree), 5 (somewhat agree), 6 (mostly agree), and 7 (strongly agree). By erring by slightly more than one point, it means that on average the prediction model could, for example, mostly agree that a given room is difficult while the human rater strongly agrees that the room is difficult.

More details about the classification results are provided in the confusion matrices shown in Figure 10.2. If the predictions were perfect, the squares in Figure 10.2 would be yellow across the diagonal. By observing the light-colored squares across the

diagonal, one notices that difficulty is the easiest criterion to predict, followed by enjoyment, and then visual aesthetics. Most of the prediction errors for both visual aesthetics and enjoyment come from predicting a score of 5 when the actual rating is a 3, 4, 6, or 7 (see the row for score 5 in Figure 10.2 (a) and (c)). This is because the score of 5 is the most common score in the dataset, being approximately 50% more common than the next most common rating. As for difficulty, the most common prediction error comes from predicting a score of 7 when the actual rating is a 4, 5 or 6, and from predicting a score of 2 when the actual score is a 1, 3, and 4.

10.3.3 Previous Neural Network Model

The problem of predicting the player's affective response was previously approached by Guzdial et al. [71]. They used a convolutional neural network to predict the responses based on direct observation of the rooms. However, the results shown in Table 10.1 and Figure 10.2 are not directly comparable to the results of Guzdial et al. This is because, even if the original data set is the same, they are comparing to averaged ratings whereas I am comparing to the non-averaged ratings (e.g., Rater A rates a room with 1, Rater B rates it with a 3—the multinomial compares to both those points, while they merge it to a single rating of 2). As a point of comparison, I ran a linear regression with the metrics selected by the multinomial LASSO regression, the results of which can be see in Table 10.2.

We see that a small number of high quality metrics can substantially outperform more advanced neural network approaches like those of Guzdial et al. by 50%.

Criterion	Accuracy	MAE
Difficulty	37.6%	1.16
Visual Aesthetics	33.1%	1.29
Enjoyment	35.2%	1.18

Table 10.1: Percentage of correctly classified rooms (accuracy) as well as the Mean Absolute Error (MAE) for the different metrics from the multinomial LASSO regressions. A more detailed analysis can be seen in Figure 10.2.

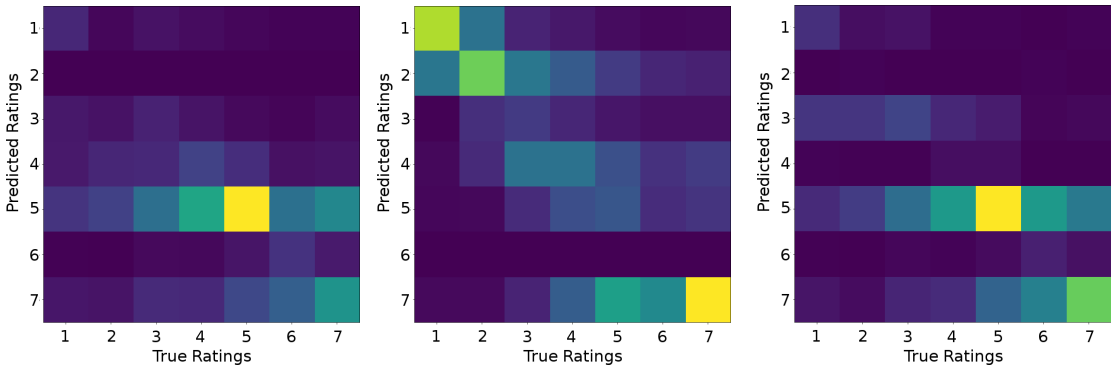
Criterion	LASSO MAE	Convolutional NN MAE
Difficulty	**0.66**	0.92
Aesthetics	**0.71**	1.13
Enjoyment	**0.68**	1.04

Table 10.2: MAE results of the linear regression using the metrics selected by the LASSO multinomial regression as input features (LASSO MAE) compared to the MAE results of Guzdial et al.'s [71] convolutional neural networks.

For example, when predicting *difficulty*, the LASSO linear regression has a MAE of 0.66 (which means that the predicted value is, in average 0.66 points off the average score provided by humans in a 7-point Likert scale), while the error reported by Guzdial et al.'s approach was 0.92.

10.3.4 Metrics Selected by LASSO

The metrics selected for each regression can be seen in tables 10.3, 10.4, and 10.5. Many of the metrics discussed do not appear in the table because they either do not provide relevant information for the task of predicting human ratings in LASSO's model, or because the information they provide is made redundant by a more informative metric. However, it is possible that metrics not selected by the LASSO could be relevant to more complex models. Prior to the regression all metrics are scaled to have a mean of 0 and variance of 1, so as to guarantee that the weights are on a similar scale. The



(a) Aesthetics (b) Difficulty (c) Enjoyment

Figure 10.2: Confusion matrices for the multinomial LASSO predictor, normalized by the number of ratings per category. Perfect prediction would be yellow along the diagonal. Difficulty is the easiest to predict with most of the errors coming from incorrectly predicting 2 and 7 too often. For both Aesthetics and Enjoyment the most common error source is predicting a 5. This comes from the fact that 5's are the most prevalent rating, being roughly 50% more common than the next most common rating, and over twice as common as most of the other ratings

Difficulty

Metric	**Weight**	ρ
Number of Enemies	1.00	0.72
Enemy σ_x	−0.07	0.50
Enemy Indicator	−0.02	0.38
Enemy σ_y	0.02	0.48
Jump Count	−0.02	−0.20
Pipe Top μ_y	−0.01	−0.20
Enemy Sparsity	−0.01	0.27
Bullet Bill σ_y	< 0.01	0.01
Path %	< 0.01	−0.12
Pipe σ_x	< 0.01	−0.20
Pipe Top %	< 0.01	−0.22
Bullet %	< 0.01	−0.02
Human Rater	ρ	
Same User	0.75	
Independent Users	0.80	

Table 10.3: The metrics selected by the Difficulty LASSO regression. The weights listed are scaled such that the maximum absolute value is 1.00. For each of the metrics, the Spearman rank coefficient is listed.

Aesthetics

Metric	Weight	ρ
Power up μ_x	1.00	0.23
Reachability	−0.58	−0.20
Number of Enemies	0.54	0.22
Negative Space	0.29	0.20
Balance	0.28	0.20
Enemy μ_x	0.27	0.17
Enemy Sparsity	0.22	0.16
Power up μ_y	0.13	0.23
Enemy Indicator	0.09	0.18
Symmetry	0.07	0.19
Bullet Bill Column %	0.05	0.06
Pipe μ_x	0.04	−0.04
Enemy σ_y	0.02	0.17
Power up Indicator	0.02	0.23
Decoration %	0.02	0.16
Density	0.01	0.13
Human Rater	ρ	
Same User	0.55	
Independent Users	0.38	

Table 10.4: The metrics selected by the Aesthetics LASSO regression. The weights listed are scaled such that the maximum absolute value is 1.00. For each of the metrics, the Spearman rank coefficient is listed.

weights listed in the table are scaled such that the weight with the highest absolute value is scaled to 1. Also listed are the non-parametric Spearman correlation coefficients, ρ, of each metric selected as relevant by LASSO's multinomial regression.

In addition to the correlation between the selected metrics and the human ratings, tables 10.3, 10.4, and 10.5 show the correlation of two independent volunteers ("Independent Users"), and the correlation between the ratings of a given volunteer ("Same User"). The correlations of Independent Users can be seen as an empirical upper bound on the expected correlation. Admittedly, the inter-rater correlation is only for two users, but even this limited subset helps set expectations on how well a

Enjoyment		
Metric	Weight	ρ
Number of Enemies	1.00	0.42
Enemy Sparsity	0.24	0.27
Power up Indicator	0.22	0.25
Power up μ_y	0.20	0.25
Power up μ_x	0.16	0.25
Negative Space	0.15	0.26
Symmetry	0.12	0.27
Enemy Indicator	0.06	0.29
Reachability	−0.05	−0.12
Enemy μ_y	0.04	0.13
Enemy σ_y	0.04	0.32
Enemy μ_x	0.03	0.24
Coin μ_x	0.03	0.16
Bullet Column σ_y	0.01	0.06
Human Rater	ρ	
Same User	0.64	
Independent Users	0.45	

Table 10.5: The metrics selected by the Enjoyment LASSO regression. The weights listed are scaled such that the maximum absolute value is 1.00. For each of the metrics, the Spearman rank coefficient is listed.

computer can achieve this task.

Figure 10.3 shows a closer look at the correlations between some of the individual metrics and the rated feature. Each plot in Figure 10.3 shows the human ratings in the x-axis and the metric values in the y-axis. Should a metric have a perfect positive correlation ($\rho = 1$) with one of the evaluated criterion, we would observe dark squares across the secondary diagonal. Similarly, should a metric have a perfect negative correlation ($\rho = -1$) with one of the criterion, we would observe dark squares across the main diagonal. The clearest trends can be seen between difficulty and Number of Enemies, where a clear linear trend can be seen. Clear trends can also be seen between enjoyment and Number of Enemies and between difficulty and Enemy σ_x.

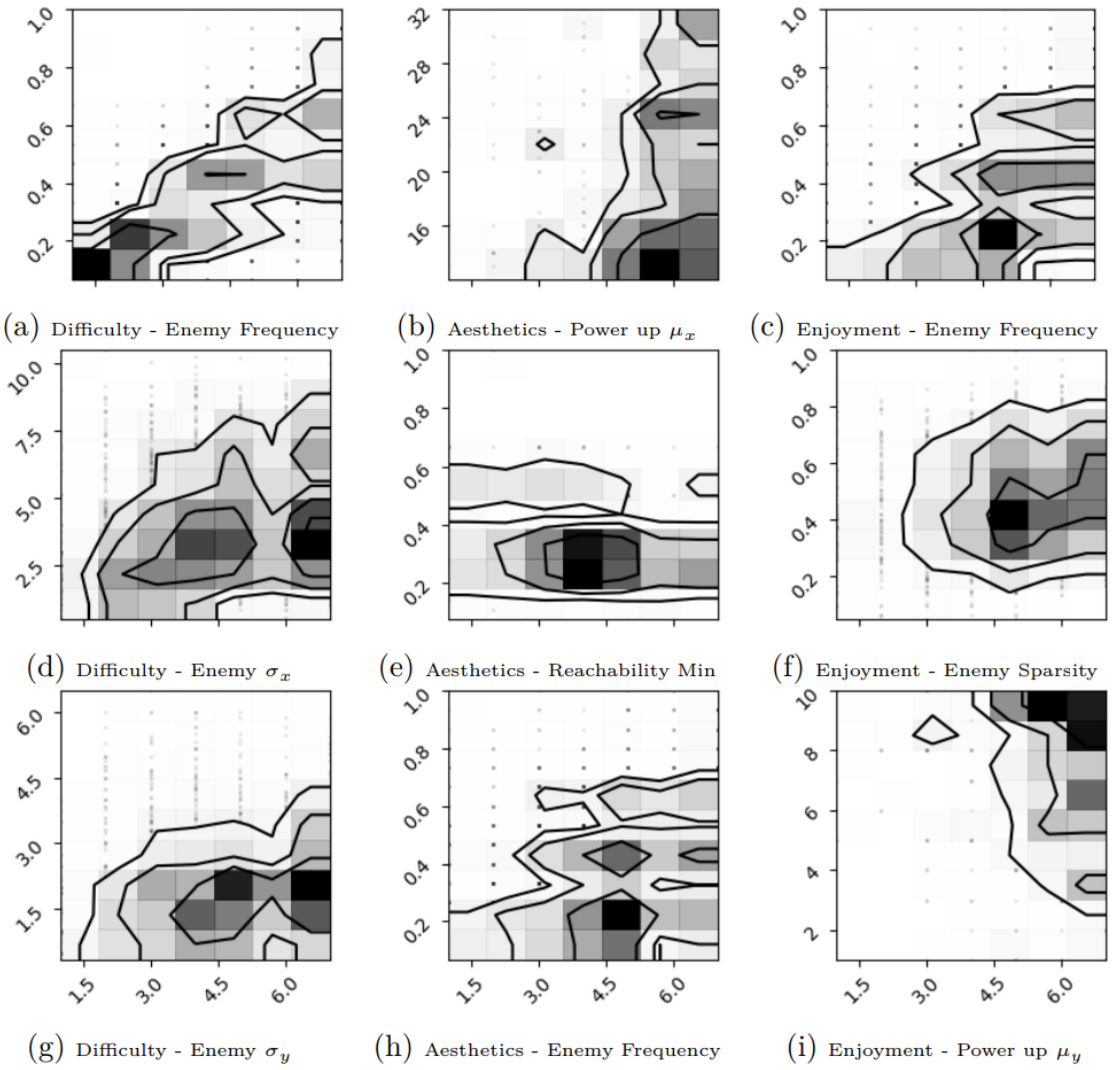


Figure 10.3: Sample density contours for the given metrics vs. the rated feature. The clearest trend can be seen in Difficulty-Enemy Frequency.

10.3.5 Discussion

The inter-rater correlations defined by the correlations between the ratings of two independent volunteers ("Independent Users") suggest that it is easier to predict difficulty, than enjoyment and visual aesthetics: the correlation between the ratings of independent users is 0.80 for difficulty, 0.45 for enjoyment, and 0.38 for visual aesthetics.

This means that different people tend to agree more in terms of difficulty, than for enjoyment or visual aesthetics. A similar trend is observed in the correlation values obtained by the best performing metrics in each criterion: the best performing metric for both difficulty and enjoyment is Number of Enemies, with correlation values 0.72 and 0.42, respectively, and for visual aesthetics it is a set of metrics related to power ups: Power up μ_x, μ_y, and Indicator all with correlation values of 0.23.

Overall, the best performing metrics in each criterion are near the inter-rater correlation given by the correlation of independent volunteers. Namely, Number of Enemies yields a correlation value nearly equal to the correlation value of the two independent users for enjoyment (0.42 for the former and 0.45 for the latter). The same Number of Enemies yields correlation value of 0.72 for difficulty, which is near the correlation value of 0.80 presented by the two independent users. Visual aesthetics is the only criterion for which the difference between the correlation of the best performing metrics and the correlation of the two independent users is larger; the best performing metrics yield a correlation value of 0.23, while the ratings of the two independent users have a correlation of 0.38.

I conjecture that the correlation values for visual aesthetics tend to be smaller because visual aesthetics is perhaps the most subjective of the three criteria. This conjecture is supported by the correlation value of only 0.55 for the visual aesthetics evaluations provided by the same person (see Same User in tables 10.3, 10.4, and 10.5).

It is interesting to note that, in contrast with the other criteria, the correlation of the ratings of two independent volunteers is slightly higher than the correlation of

the ratings given by same volunteer for difficulty—0.80 for the former and 0.75 for the latter. A possible explanation is that multiple scores given by the same volunteer for a fixed room are subject to ordering effects. That is, a room will likely be easier the second time a person plays that room.

10.3.5.1 Difficulty.

The metric that obtained the highest correlation was Number Enemies, which simply counts the number of enemies in the room. The metric of leniency (not shown in tables 10.3, 10.4, and 10.5 because it was not selected by LASSO) obtained a correlation of 0.53, which is much lower than the correlation obtained by the simpler Number of Enemies. Thus Number of Enemies is the current state-of-the-art single metric for predicting human-perceived difficulty in the dataset used in the experiments. However, this might just indicate that difficulty in rooms generated by the Notch Level Generator used in the experiments comes primarily from adding enemies, and not by other factors such as gaps or platform configurations.

The number of enemies is the most important metric by over a factor of 10 when considering the regression weight but is not the only important metric. Moreover, notice that when interpreting the results presented in tables 10.3, 10.4, and 10.5, have in mind that the correlation coefficient ρ is calculated for each metric individually, but the regression weight results from applying LASSO to all metrics at the same time. So, a low regression weight might not mean that a metric is not relevant, but could also mean that LASSO found other metrics that represent similar information, and thus did

not have to assign a higher weight to a given metric.

We see that the horizontal spread of the enemies (Enemy σ_x) has a negative impact on the difficulty (negative LASSO regression weight), meaning that humans tend to find rooms with a larger spread of enemies to be easier than rooms with a smaller spread of enemies. This is interesting because the horizontal spread has a positive correlation with difficulty. It is possible that this sign discrepancy between LASSO's weight and correlation value for Enemy σ_x happens because the metric acts as a proxy for number of enemies when analyzed individually (correlation value). By contrast, when analyzed altogether with metrics that already account for the number of enemies (e.g., LASSO regression also accounting for Number of Enemies), Enemy σ_x shows that humans tend to find easier to play rooms in which the enemies are spread out. Intuitively, it makes sense that a larger spread is easier since the enemies will be spaced out, while a dense cluster will present a more difficult obstacle. Conversely, the enemy vertical spread (metric Enemy σ_y) has a positive effect on the difficulty. Again, this makes sense as a large vertical spread of enemies tightly clustered horizontally will present a wall of enemies that is hard to navigate, whereas a tight cluster vertically can be avoided.

10.3.5.2 Visual Aesthetics.

The metric of Symmetry is amongst the best performing metrics for visual aesthetics with respect to correlation values ($\rho = 0.19$). Note, however, that one might have expected a negative correlation between S-values and visual aesthetics. That

is, symmetrical rooms (small S-values) to be rated as visually pleasing by humans. However, the opposite is indicated here: the positive correlation for Symmetry and visual aesthetics means that rooms with larger S-values (less symmetrical rooms) are rated as more visually pleasing.

The explanation for this discrepancy is rooted at the number of objects in the rooms: Symmetry might be working as a proxy for Negative Space. Intuitively, rooms with fewer objects tend to have much smaller S-values than rooms with a large number of objects. This is because with more objects the values of X_M, Y_M, and A_M tend to be larger (see Equation 10.1 and the definitions of X_M, Y_M, and A_M). As an example, the S-value of an empty map is trivially zero. The positive correlation for Symmetry is explained by the fact that human subjects tend to attribute low visual aesthetics scores to rooms with very few objects, and high scores to rooms with more objects—in this study the symmetry metric is essentially measuring how much of the grid is filled with objects. Therefore, it is not surprising that Negative Space and Symmetry yield similar correlation values: 0.20 for the former and 0.19 for the latter.

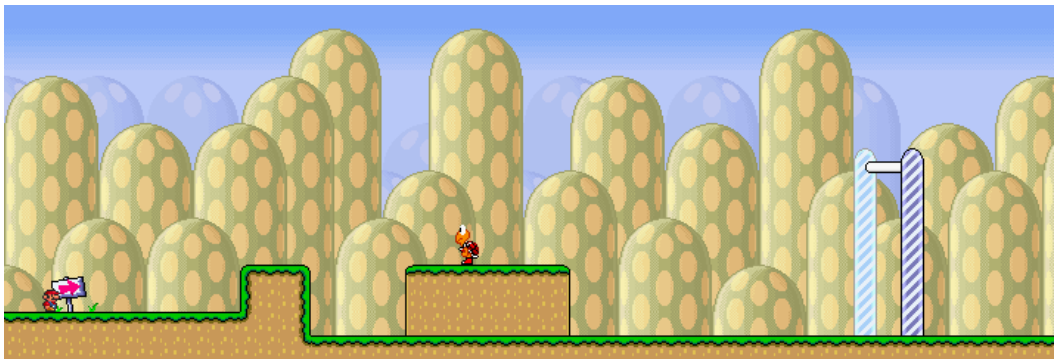
Other metrics that show very strong correlations are related to the presence of power ups (e.g., Power Up $X\mu$ and Power Up $Y\mu$), perhaps because power ups are usually scarce and could appear amongst other more elaborate decorative tiles. The somewhat strong correlations between Negative Space and Reachability with visual aesthetics suggest that reachable vertical variety tends to be appreciated by the player. Interestingly, LASSO attributed small weights to both Power up Indicator (weight 0.02) and Power up Frequency (weight of 0.00, as the metric does not appear in tables 10.3,

10.4, and 10.5) and much larger weights to Power up μ_x and Power up μ_y. This difference in the weight values suggest that it is not the mere presence of power ups (Power up indicator) or the amount of power ups (Power up Frequency), but the positioning of the power ups that is most important for the aesthetics. The further to the right and higher the power ups (i.e., large values of both Power up μ_x and Power up μ_y), the more aesthetically pleasing it was to the players. Perhaps players find it aesthetically pleasing to have a reward towards the end of the room that requires maneuvering to reach and dislike being handed a power up at the beginning.

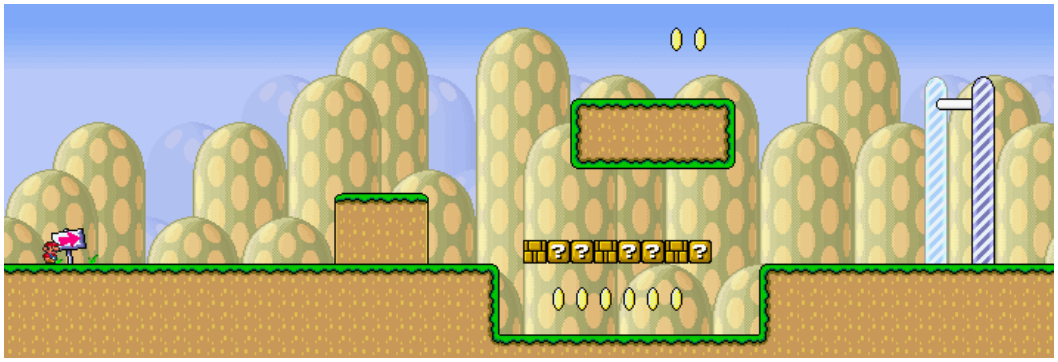
We also see Number of Enemies appears as a well performing metric speaking to the fact that players enjoy the variety that enemies bring. Interestingly, enemies are the only such type that have this effect. As discussed above, players find the presence of power ups pleasing, but the amount has a minimal effect. The amount of Bullet Bill Columns has an effect, but perhaps interestingly, not the amount of Bullet Bill cannons. This means that the larger the column, the more visually pleasing, but that adding more cannons does not necessarily improve the human-perceived visual aesthetics.

10.3.5.3 Enjoyment.

The metrics that correlated the most with enjoyment tend to be metrics related to elements in the game that the player can interact with (enemies, power ups, and coins). This is related to the Yerkes-Dodson law [225] demonstrated by Piselli et al. [147] in the context of video games. According to the Yerkes-Dodson law, enjoyment will be maximum for the right amount of challenge. The strong correlation between Number



(a)

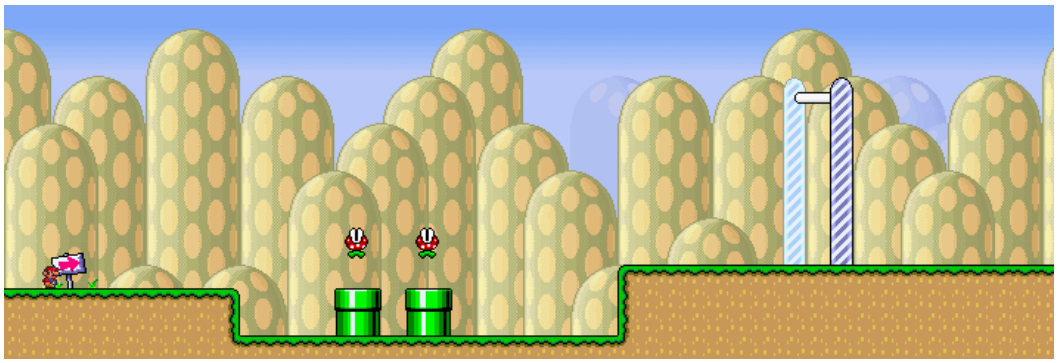


(b)

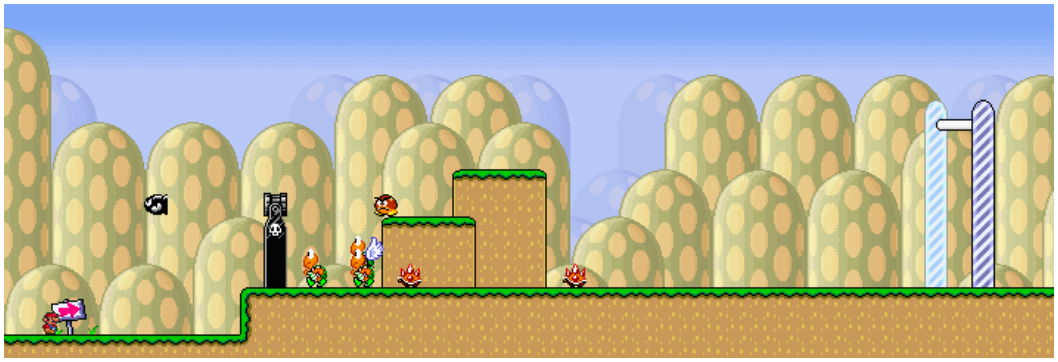
Figure 10.4: Two rooms with high misclassification error for the Difficulty rating. Both were classified as 1 by LASSO, but both have ratings of 7. Given that the players who rated them as 7's had no difficulty completing the room, we believe that it comes from a misunderstanding of the rating scale.

of Enemies and enjoyment ($\rho = 0.42$) suggests that the right amount of challenge for Reis et al.'s volunteers included a large number of enemies. As mentioned above, a wide spread of enemies vertically indicates more challenge for the player and the fact that it was one of the highest correlations ($\rho = 0.32$) reinforces that this challenge is enjoyable for players.

Interestingly, while the counting-based metric Number of Enemies is the most important factor for enjoyment (LASSO weight of 1.00 and ρ of 0.42), positioning-based



(a)



(b)

Figure 10.5: Two rooms with high misclassification error for the Difficulty rating. Room (a) was classified as a 1 by LASSO but was given a rating of 6 by a player, while Room (b) was classified as a 7 by LASSO but was given a rating of 2 by a human.

metrics also seem to be important. The mean horizontal position for enemies (Enemy μ_x), power ups (Power up μ_x), and coins (Coin μ_x) all have a positive impact on the players' enjoyment. This seems to indicate that players enjoy a brief amount of respite at the beginning and appreciate higher complexity towards the end of the room, which is a common room design tactic [215].

Additionally, metrics concerning the distribution of platforms such as Symmetry and Negative Space also had a high correlation. These metrics reflect the type of movement that players can execute through rooms, again reinforcing that enjoyment

is linked to how the player interacts with the room. Furthermore, Reachability has negative weight and ρ values, suggesting that the player finds rooms containing objects that they cannot interact with less enjoyable.

The metric with largest LASSO weight after Number of Enemies is Enemy Sparsity (weight of 0.24). Similar to Enemy σ_x, Enemy Sparsity also computes the spread of enemies in the room. The positive weight and ρ values for Enemy Sparsity in enjoyment indicate that people tend to find rooms in which the enemies are spread out to be more enjoyable. Although the difference between Enemy σ_x and Enemy Sparsity is subtle (the former returns larger values than the latter for rooms with enemies too far from the average enemy position), the results suggest that this subtle difference is important. That is, if Enemy Sparsity is removed from the pool of metrics, LASSO selects 24 instead of the 14 metrics shown in tables 10.3, 10.4, and 10.5. This increase in the number of selected metrics suggests that one needs approximately 10 other metrics to make up for the lack of Enemy Sparsity.

10.3.6 Case Studies

To further address some inconsistencies, consider a few of the room snippets that had the highest misclassification error for Difficulty. Difficulty is chosen since (1) it has the highest inter-rater reliability so we are more likely to able to make valid judgments and (2) the metrics and regressions both have the best predictive power for difficulty so disagreements are probably fundamental, and not a factor of noise.

In Figure 10.4 shows two rooms that showcase the difficulty of the prediction

task. Both of these rooms were predicted to be a 1 in difficulty by LASSO, i.e., the easiest rooms possible. However, human raters attributed a difficulty score of 7 to both of them, i.e., the hardest rooms possible. At a glance, one can tell that these rooms are indeed easy, with either a single, easily dodged enemy (a) or no possibility for death (b). In this case, it is likely that there was a misunderstanding of the rating scale, i.e., the volunteers thought that 7 was easy and 1 was most difficult, or that the raters were not performing the task faithfully. In both cases, other raters rated the rooms as extremely easy, giving them 1's or a 2, in the case of (a).

Figure 10.5 shows two rooms that were incorrectly classified for legitimate, interesting reasons. The room shown in Figure 10.5.(a) has two piranha plants in the middle of the screen. A patient player can bide their time, wait for the plants to return to the pipes and continue on their way. However, a novice player who is unaware of how the piranha plants behave could easily be in mid-jump over a pipe when the piranha plant emerges, catching them by surprise and killing them. In fact, the player who rated the room as a 6 in difficulty died on the room, so it is likely that they were caught by surprise. While from a purely count based view, the room is easy, hence why it was classified to be a 1, it offers enough surprise that a novice player could find some difficulty with it. The LASSO regression incorporates no knowledge about player familiarity or skill, which may not be representative.

The room shown in Figure 10.5.(b), has a dense cluster of enemies. At first glance, this appears to be an intimidating, skill intensive block for players. However, the goomba is about to fall off of the higher platform, leaving a clear path for a patient

player who hops up to the bullet bill cannon and jumps over to the now clear path over the enemies. While one of the raters did die on this room, rating it a 5 instead of the 6 that we classified it as, the other passed it with no trouble bypassing the enemies altogether. Again, a novice or intermediate player is likely to have difficulty either through nervousness or a desire to kill all of the enemies (the player who died killed 7 enemies in total) that gets them in trouble, while the advanced player just ignores the enemies. Thus, a count-based metric can only find that a tight cluster of enemies is correlated with difficulty, even if there are clear paths through the room.

10.4 Conclusion

Perhaps disappointingly, most of the proposed interesting metrics (i.e., those that are not simple baselines) hold no explanatory power over the simplest of metrics (simple counting and shear existence of entity types). However, I believe this speaks more to the fact that much more research is required to find accurate, important metrics than to the fact that counting the number of enemies in a level is a good metric. But, it also indicates that ad hoc construction of metrics is unlikely to bring about the desired result in the qualitative analysis of the expressive range of a generator (i.e., what does it mean to "Have good coverage of a metric space," if that metric is a poor indicator of any human affective response). However, even seemingly inconsequential metrics can be important for quantitative comparison of a machine learned generator and its input corpus – as a good machine learned generator should be similar to its input corpus in

all metrics, even if those metrics carry no other weight. The next chapter describes techniques for the comparison of generators' generative space to that of their training corpus, both qualitatively and quantitatively.

Chapter 11

Analyses of Generative Space

While the choice of metric can be difficult, as discussed in the previous chapter, the analysis of a generator in metric space is still valuable. While the actual content generated is the single most important factor, it is nigh impossible to assess the quality of every piece of content (or even a large number) generated. Thus, metrics are a necessary evil, and while any single metric (or set combination of metrics) is possible to be hacked by a given generator, by comparing generators across a wide range of metrics, this is diminished. For classical PCG approaches based on search [213] that are guided by the optimization of a specific criterion (or weighted sum of criteria), selecting or creating the correct metrics can be a difficult task. If the chosen criterion is a poor metric, then the resultant content is unlikely to meet the expectations of a designer. [1]

In this way, machine learned approaches have it easy. They do not need to

[1] Portions of this chapter originally appeared in "Super Mario as a String: Platformer Level Generation via LSTM"[197] and "Studying the Effects of Training Data on Machine Learning-Based Procedural Content Generation" [**?**]

optimize for a given metric, or, rather, their metric is to produce the model that is best capable of generating a piece of content like the content from the input corpus. Similarly, they have a more clearly defined version of success than a classical PCG system. Instead of optimizing an ad hoc metric, a PCGML system's goal is to produce a generative distribution within which the input corpus could likely be found. As such, on any given metric, the generator should produce a similar distribution to that of the training corpus. Moreover, it is not enough to find a generator that is similar to that of the input corpus on the marginalized distributions, as it is the joint distribution of all metrics that needs to be matched. Given the fallibility of humans in selecting good metrics (as evidenced in the previous chapter where the simplistic), the goal should not be to target any one or two metrics, but instead to consider the generative space holistically, with every imaginable (of course limited due to time and feasibility constraints) metric considered at once.

In the following sections, I will discuss methods for analyzing generators holistically, first qualitatively in an extension of expressive range analysis, then discussing ways to detect plagiarism in a generator, then on how to quantitatively analyze how close a generator is to the target distribution, and finally a discussion of how to best showcase results of a generator.

11.1 Kernel Density Estimation and Visual Inspection

Kernel Density Estimation (KDE) [155] is a statistical method for estimating the probability density function of a distribution from a set of sampled data. In the generation process of a PCG system, the individual pieces of content are independent (i.e., the sampling of the third piece of content is not dependent on the first or second) and identically distributed (i.e., they come from the same generative process) from an unknown distribution, even if a generator has perfect knowledge about some metrics of distribution, there can always be a new metric for which it does not know the underlying distribution. The heat density plots of Smith and Whitehead were an attempt to analyze these distributions; however, KDE seeks to find analytic functions that best describe the sampled results. The result is a cleaner system to determine the actual generative space of a PCG system, and a way to more accurately compare two generators.

Given $(x_1, x_2, ..., x_n)$ samples, then the kernel density estimator of the true probability density function, f, is:

$$\hat{f_H}(x) = \frac{1}{n} \sum_{i=1}^{n} K_H(x - x_i)$$

where H is a positive definite symmetric bandwidth matrix of the KDE, and K is the multi-variate kernel. The bandwidth is a hyper-parameter of the KDE that acts to smooth the supplied data points. As $H \to \infty$ the KDE become smoother and smoother, loosing all shape. Conversely, as $H \to 0$ the KDE becomes closer and closer to the supplied data. The selection of the bandwidth is an open area of research [55, 56]. In this work, I use the multivariate plug-in bandwidth selection with unconstrained pilot

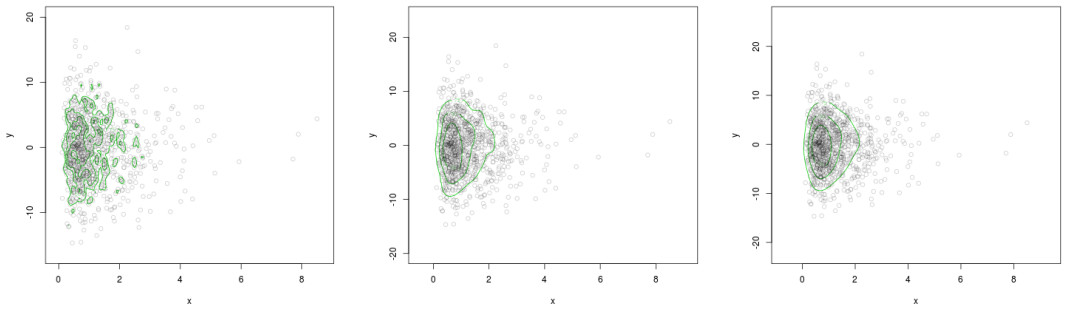
bandwidth matrices approach of Chacón and Duong [35] to find a suitable bandwidth. The approach seeks to find the bandwidth matrix H that minimizes

$$\mathbb{E} \int_{\mathbb{R}^d} (\hat{f_H}(x) - f(x))^2 dx$$

the mean integrated square error of the smoothed density and the true density. However, since the true density is unknown, in general this is approximated by the asymptotic mean integrated square error

$$\text{AMISE}(h) = \frac{R(K)}{n}|H|^{-1/2} + \frac{1}{4}m_2(K)^2 H^4 R(f'')$$

where $R(K) = \int K(x)^2\,dx$ and $m_2(K) = \int x^2 K(x)\,dx$.



(a) Scaled by $\frac{1}{10}$. (b) Unscaled (c) Scaled by 2.

Figure 11.1: Comparison of three different scalings of the bandwidth matrix as calculated via the multivariate plug-in bandwidth selection with unconstrainted pilot bandwidth matrices approach of Chacón and Duong [35]. The left figure shows a bandwidth that is too small, leading to an coarse estimation. The figure on the right is beginning to lose the shape along the x axis.

A visualization of how the bandwidth affects the estimation can be seen in figure 11.1. In general, while most of the downstream processes are not particularly sensitive to the choice of bandwidth – most notably the calculation of expressive volume discussed in section 11.2 – the approach of Chacón and Duong bandwidth will be used for all following density estimates.

11.1.1 Comparison to the Original Corpus

Expressive range analysis is a qualitative visual assessment of metric space of a generator. While this is a good practice in general, it is especially important for a machine learned generator, but only in so far as a comparison tool between the metric space of the generator and that of the original corpus. The goal of a PCGML system is to match the metric properties of the original corpus – the design latent within. While this qualitative analysis is insufficient for rigorous comparisons between generators, it allows for exploratory understanding of the type of content producible by a generator – and ways in which the generator is biased away from the type of content found in the original corpus.

The classic two dimensional histogram used by Smith and Whitehead (as shown in figure 9.1) is poorly suited to the evaluation of the metric space of the rooms of most games. Smith and Whitehead focused on the analysis of generators that could easily create thousands of pieces of content, for which the histogram acted as a good estimate of density. However, most games tend to have a relatively small number of rooms – on the order of a few dozen – (although a few reach into the thousands, e.g. *N++*), which leads to extremely sparse histograms, hence the need for density estimation. The histogram for the above and below ground rooms from *Super Mario Bros.* with the same number of bins as figure 9.1 can be seen in figure 11.2. Note that only one cell has more than one data point.

This is why kernel density estimation is such an important tool for PCGML

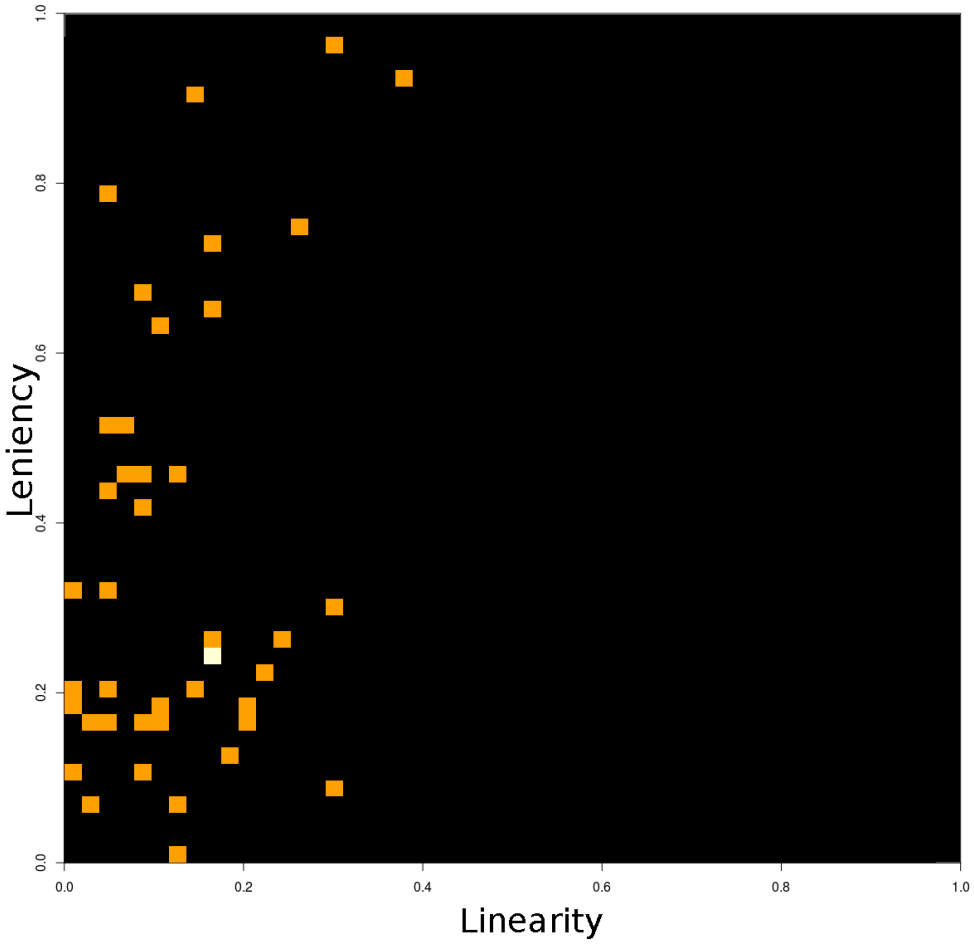


Figure 11.2: Expressive range plot of the above and below ground rooms of *Super Mario Bros.* Note that only one cell has more than one data point (the white data cell).

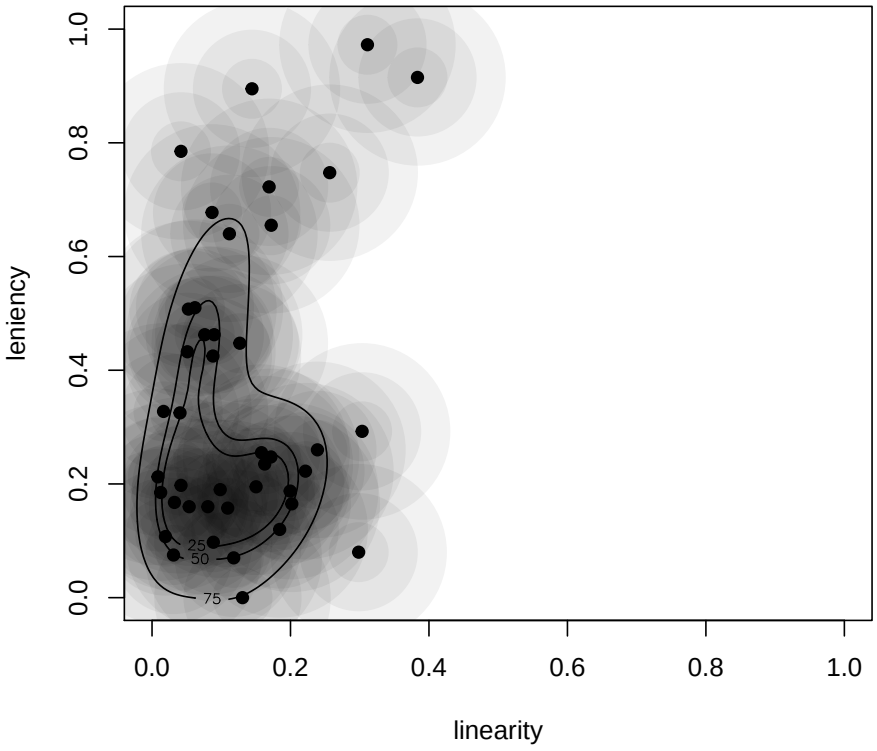


Figure 11.3: Density estimate for *Super Mario Bros.* The contours represent the $25^{th}, 50^{th}$, and 75^{th} percentiles of the estimate. The KDE makes it much easier to see the pear shaped relationship between linearity and leniency in *Super Mario Bros.*

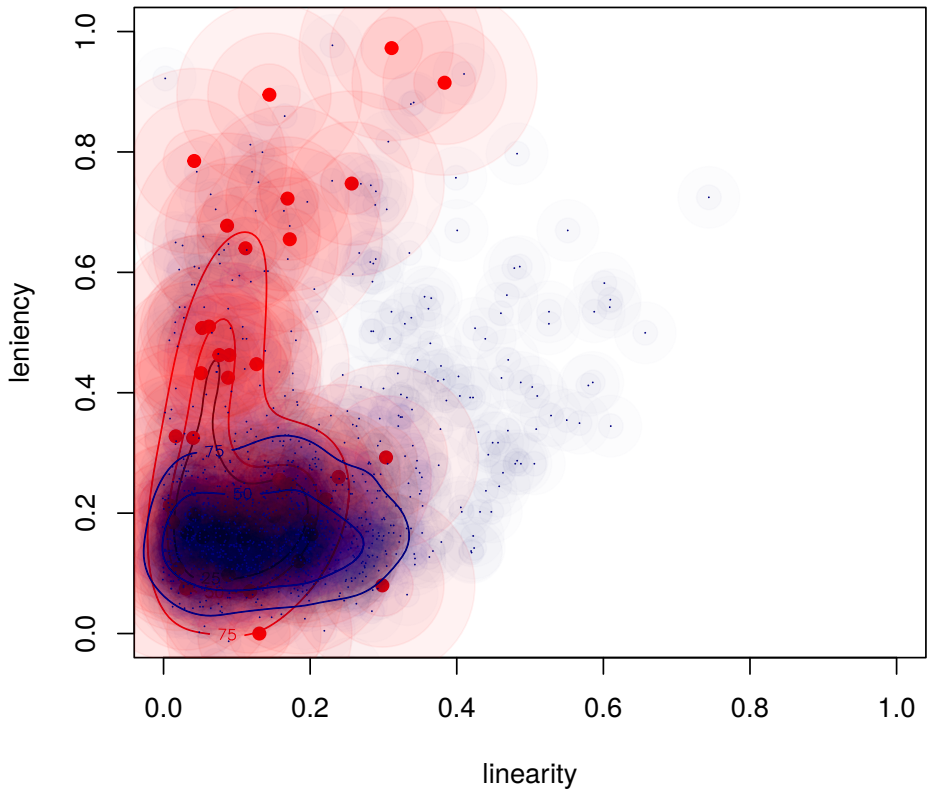


Figure 11.4: Density estimates for *Super Mario Bros.* (Red) and the 512 unit, 80% dropout *Snaking* with depth and path information generator (Blue).

based generators. As humans, it is difficult to estimate a density function from visual inspect of a sparse histogram. For comparison, the calculated density for the linearity and leniency of the input corpus from *Super Mario Bros.* can be seen in figure 11.3. While a useful visualization tool in and of itself, it is in comparison that it shines. Figure 11.4 shows a comparison between the 512 unit, 80% dropout *Snaking* with depth and path information generator and the original rooms. The regions of highest density (between 0 and 0.2 for both linearity and leniency) for both have a high degree of overlap, but the original levels have some rooms with higher leniency than *Kamek* seems capable of supporting. Conversely, *Kamek* produces levels that are much more linear than those of *Super Mario Bros.*. While *Kamek* is putting too much support in the highly linear region, it speaks to its generalization that it is capable of producing rooms outside the scope of the original set. The lower edge of the fan shape of *Kamek* has a similar trajectory to the pear shape, speaking perhaps to an inherent relationship between linearity and leniency – holding all other factors constant, a more linear level is going to be more lenient (i.e., it has fewer jumps required to complete it).

To further this comparison, figure 11.5 adds in *Kamek* generator with 64 hidden units. We see that this generator places more support in the middle to high leniency values. Furthermore, it generalizes much more poorly in linearity than the larger *Kamek* model. Perhaps the largest indicator for its poor capturing of the original rooms' design is the lack of nearly any support in the region of highest density in the originals (between 0 and 0.2 for both linearity and leniency).

While this two dimensional comparison is useful, as shown in the previous

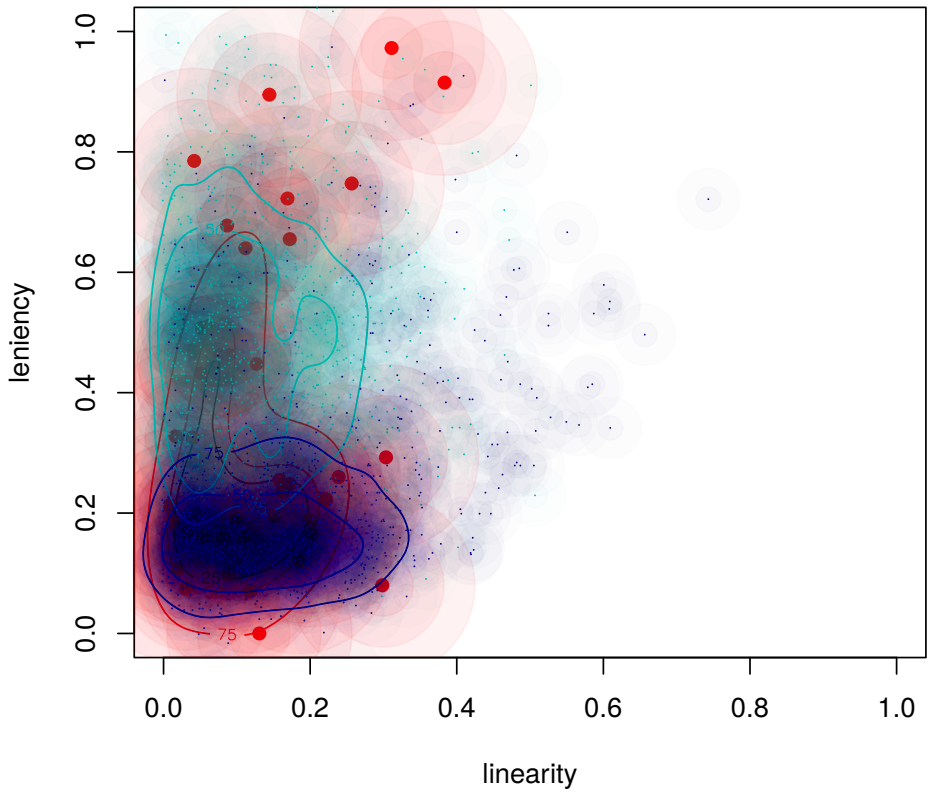


Figure 11.5: Density estimates for *Super Mario Bros.* (Red), the 512 unit, 80% dropout *Snaking* with depth and path information generator (Blue), and the 64 unit, no dropout *Snaking* with depth and path information generator (Teal).

chapter, there are a large number of metrics that one might wish to explore. While *Danesh* allows for the switching of what metrics are viewed, this interaction pattern makes it difficult to discern higher dimensional structure in the generative space. While a three dimensional plot can be navigable in an interactive setting [156], it is difficult to discern the the structure via a static image. Corner plots [61] are a visualization technique that allows for an arbitrary number of dimensions to be viewed holistically. Figure 11.6 shows a comparison of the same sets as in 11.4, but across a much larger set of metrics. The metrics shown here are:

- e – The frequency of the room taken up by empty space
- n – The negative space of the room, i.e., the percentage of empty space that is actually reachable by the player
- d – The frequency of the room taken up by "interesting" tiles, i.e., tiles that are not simply solid or empty
- p – The frequency of the room taken up by the optimal path through the room
- l – The leniency of the room, which is defined as the number of enemies plus the number of gaps minus the number of rewards
- R^2 – The linearity of the room, i.e., how close the room can be fit to a line
- j – The number of jumps in the room, i.e., the number of times the optimal path jumped

- j_i – The number of meaningful jumps in the room. A meaningful jump is a jump that was induced either via the presence of an enemy or the presence of a gap.

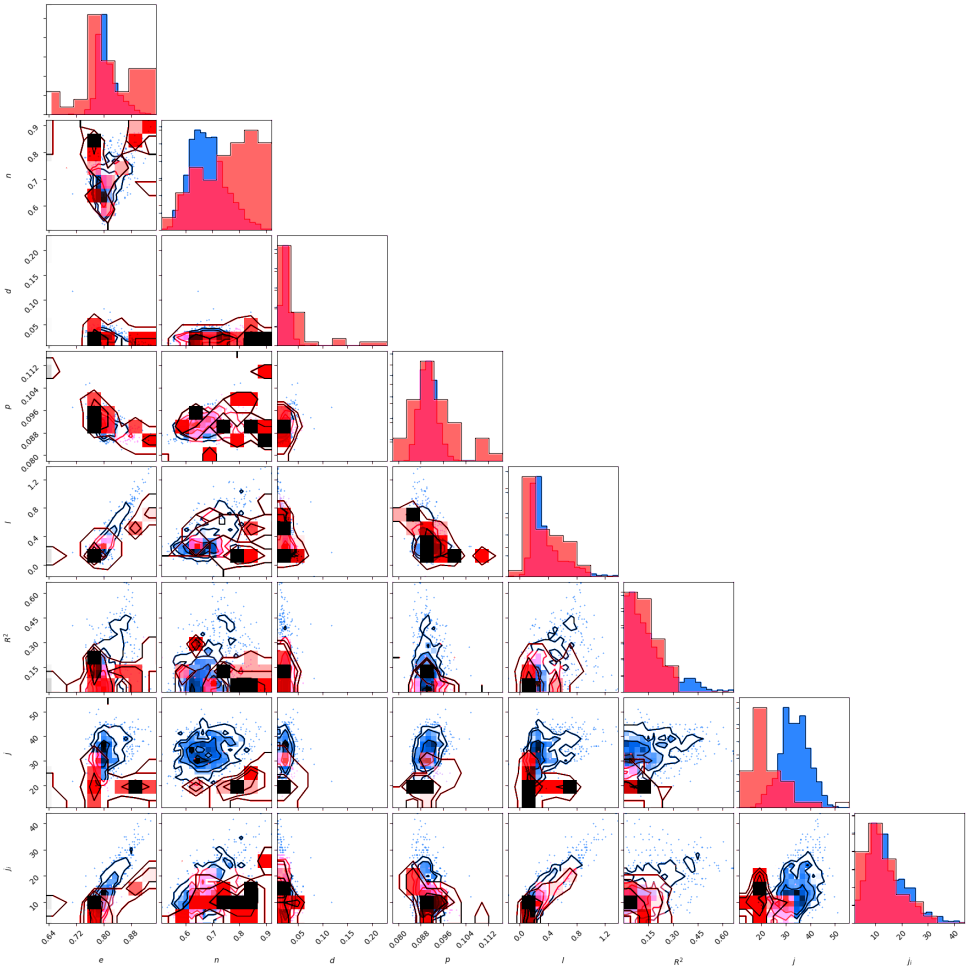


Figure 11.6: Corner plot for *Super Mario Bros.* (Red) and the 512 unit, 80% dropout *Snaking* with depth and path information generator (Blue)

Looking closely at figure 11.6, we can tease apart where *Kamek* does a good or bad job of capturing the metric space of the original rooms.

On the uni-dimensional analysis (i.e., the histograms that make up the diagonal of the corner plot):

- e - *Kamek* and the originals have the same peak, but the originals have a lower amounts. This is due to *Kamek* undersampling from the below ground generative space.
- n - *Kamek* produces rooms with noticeably lower amounts of negative space. The below ground undersampling accounts for a portion of this, but even accounting for that, *Kamek* still has a very different peak (0.65 for *Kamek* vs 0.8 for the originals.
- d - A relatively uninteresting metric as most rooms are around 0.05 – due to the extremely high peak, *Kamek* does a poor job of emulating the rooms with higher frequencies of "decoration".
- p - *Kamek* has a similar peak, but a less wide distribution of path length.
- l - *Kamek* matches the peak of the originals, but in general (as discussed above) under samples the more lenient rooms.
- R^2 - The major case where *Kamek* has a wider distribution than that of originals.
- j - The metric by which *Kamek* does the worst job of matching the target distribution, producing rooms with a much higher number of jumps.
- j_i - Despite *Kamek* producing rooms with more jumps, the distribution of meaningful jumps is very close to that of the origina rooms.

Looking at the multi-dimensional cases, ignoring the uni-dimensional discrepencies, *Kamek* does a good job of capturing the relationships between the metrics. The

most notable relationships are those of the meaningful jumps, j_i, cross empty space, e, negative space, n, and leniency l. All of those have a discernible trend where the rooms with more meaningful jumps have more empty space, more negative space, and are more lenient. This relationship is also captured by *Kamek* which would seem to indicate that *Kamek* is in fact learning some of the latent design choices.

While a broader expansion of expressive range is useful as a design paradigm, it is only one way in which the generative space of a generator can be analyzed. The following section will discussion expressive volume, a single numerical value that can help in understanding the qualities of a generator.

11.2 Expressive Volume

In the literature it is common to refer to the "width" of the expressive range, but this has yet to be done in anything beyond a qualitative visual assessment [178]. However, width is problematic as it is a linear dimension. For instance, a generator that always produced perfectly linear levels that hhave a very wide range in leniency would still be unlikely to be thought of as very expressive, given that it is completely lacking in 1 dimension. To this end, the expressive *volume* is the measure that is actually desired for the understanding of a generative space. Volume is the term that will be used from here on out, which can be taken to mean the $n-$dimensional volume (e.g., area in 2D, standard volume in 3D, etc.) of the generated metric space to be the *size* of the expressive range.

Similar to expressive range, the absolute size of the expressive volume is essentially meaningless in the abstract (i.e., is it necessarily good to have a large expressive volume?), but in comparing a PCGML generator and the original data, one would expect similar volumes for a generator that is correctly learning the target distribution. Of course, volume is problematic in its own right, as two generators could sample very different portions of the generative space and have identical volumes, which is why expressive volume is insufficient as the sole technique and should be used in conjunction with other techniques.

To calculate this volume, the KDE of the generative space is used. The KDE is found via the afore mentioned method, and is then thresholded for points greater than 0, which is take to be the boundaries of the expressive range. One then forms an $n-$dimensional grid, and counts the number of bins that lie within the expressive $n-$volume, multiplying the count with the volume of a single bin. Via experimentation this volume was found to be relatively robust to the choice of grid size. A range of 50, 100, and 200 bins per dimension were used with less than 5% variation found in the relative ratio of volumes for two different density estimates (the *Kamek* 512 unit generator and the original levels).

11.2.1 Case Study – Expressive Volume as Function of Training Corpus

While PCGML systems are always going to be starved for data, an important question to answer is to what degree does the amount and quality of data affect the

generative properties. Expressive volume is used as a proxy for how well a generator captures the desired properties of the original corpus. For this experiment the expressive volume is computed using linearity, leniency, and enemy sparsity for the density estimation.

In addition to evaluating the performance of a PCGML method when using increasing amounts of training data, this experiment also evaluates using training data of varying "quality."

"Quality" is a subjective term, but as discussed in section **??** the most likely level is one that is very uniform and boring. Thus, a proxy for "quality" can be to consider levels that are more uniform to be of "lesser quality" than levels with move variety. We thus approximate quality by computing the entropy of the training levels through their high-level structures. That is, we split the training levels into 4×4 tile sections. We then perform k-medoids on those sections with $k = 30$ [100]. For the k-medoids distance metric, we find the positioning of two sections that yields the most overlap in tile types between the sections, and weight that by the area of the overlapping sections. The idea is that this metric provides us with a measure of how structurally similar two sections are. Once the clusters are computed, we represent each level as a histogram containing the number of 4×4 level sections belonging to each cluster. Finally, we compute the entropy of those histograms [220], and assume that a higher entropy corresponds to more information in that level, and thus a higher quality training level.

11.2.1.1 Training Sets

In order to evaluate the effects of both the quality and quantity of training data on the chosen models, several sets of training data are considered. We first order the training levels from most to least entropic. We then train separate models using the first 16 columns of the most entropic level, using the first 32 columns, using the first 64 columns, using the first 128 columns, using the most entropic level in its entirety, using the two most entropic levels, etc. We then repeat this process using the least to most entropic ordering of the levels. In total, 66 models are trained. Figure 11.7 shows the

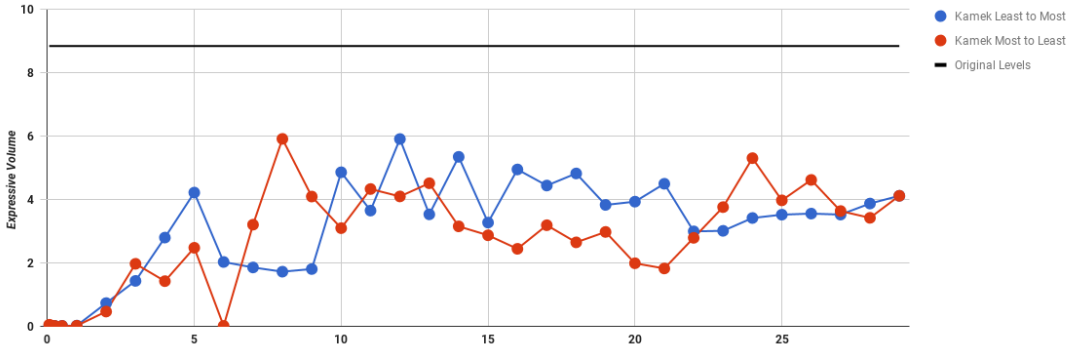


Figure 11.7: The expressive volume of *Kamek* as a function of the training data.

expressive volume of *Kamek* as a function of the amount of data for both the most-to-least and least-to-most entropic training sets. Also shown is the expressive volume of the original levels. Notice, what we care about here are the ratios between the expressive volumes of the models, as the actual volume scales will vary for different domains. In general, we see that a small amount of data ($<$ 3 levels) results in a very small expressive volume, which is as we would expect given that there isn't much variation in the supplied data. Surprisingly, additional data does not increase the expressive volume

of any of the models after about 5 training levels. All generators have a much smaller expressive volume than that of the original levels, which is 50% larger than the next closest generator (Most-to-least 8 levels). This larger volume is present in all of the individual metrics, meaning it is not just a failing in any one particular aspect. Note that after reaching 5 levels worth of data, the information density of those levels does not effect the expressive volume of the models. Perhaps, this is due to the fact that the variety found in multiple levels, even those of relatively low information content, exceeds that of any one level. Of course, there are certainly degenerate counter-examples (e.g., 5 empty levels would provide no worthwhile information), but in any reasonable practical application it is more important to acquire a sufficient amount of data than to cherry-pick the highest quality data.

Expressive volume represents a natural, quantitative follow on to the, now standard, expressive range analysis of procedural generators. In particular, it allows one to understand the "width" of the generative space via a metric that is relatively robust. However, as discussed, it is insufficient as the sole evaluative criterion, as a desirable criterion should assess not just the relative sizes of the metric spaces of generators, but also the location and shapes of these metric spaces. In the following section, I will discuss a robust multi-dimensional method for the comparison of generative spaces.

11.3 Joint Distribution Testing of Generative Spaces

Where expressive volume has failings in terms of not capturing the locality of the density function, many traditional one-dimensional tests succeed. There are a number of uni-dimensional tests for whether two samples came from the same distribution. In the case of evaluation for a PCGML, we know a priori that the samples came from different distributions, but the goal is still the same. If two samples (e.g., a sample of levels from a generator and the original levels) are believed to have come from the same distribution, then the generator has achieved its goal.

The most common of these tests is that of Student's t-test [191], a common test for whether two samples came from the same distribution. However, a key assumption of the t-test is that the underlying distributions are Gaussian Normal distributions. From visual inspection, it is easy to see that most of the metrics considered are not normally distributed. Of the metrics considered in figure 11.6 only path length, p, and number of jumps, j, might plausibly be normally distributed. Thus, the t-test is unsuitable for the general case of comparing two distributions.

While the t-test is not suited for this, there are non-parametric tests that accomplish similar comparisons when the distributions can not be assumed to be normally distributed, most notably the Mann-Whitney U-test [116]. The Mann-Whitney test compares the ranked values for two samples – i.e., all observations from two samples are ranked, and then the ranks for each sample are summed (R_1, R_2). The test statistic

is then calculated as:

$$U_1 = R_1 - \frac{n_1(n_1+1)}{2} \tag{11.1}$$

$$U_2 = R_2 - \frac{n_2(n_2+1)}{2} \tag{11.2}$$

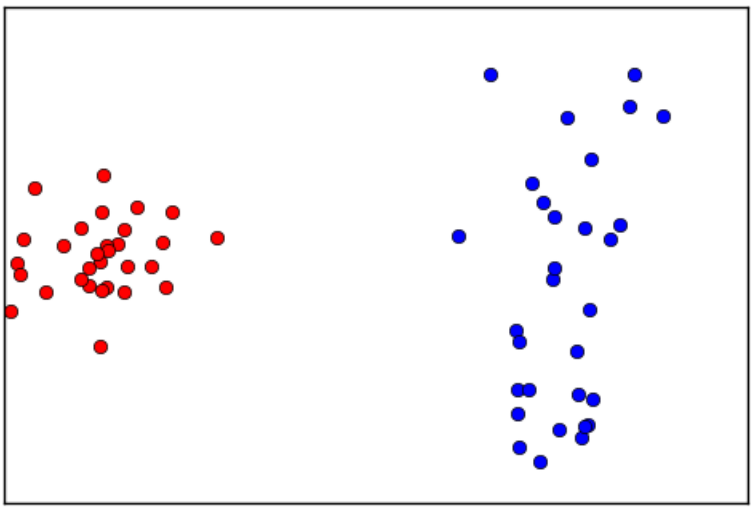


Figure 11.8: An example of the fallibility of the Mann-Whitney U-test. Shown are two samples, one from a unit Gaussian Normal distribution ($\mathcal{N}(0,1)$) and one from a uniform distribution ($\mathcal{U}(-5,-5)$) (note: x-axis used only for visual separation). While these distributions are obviously different – even to visual inspection – the Mann-Whitney test is unable to reject the null hypothesis of both distributions being equal ($U = 380, p = 0.15$)

.

where n_1 is the number of observations in sample 1, and n_2 is the number of observations in sample 2. To calculate the p-value for the test, the smaller of the two is used as the test statistic. However, since it only compares ranks, the test relies on the assumption that the two distributions will both have identical shape and scale for the test to be valid. Figure 11.8 shows the flaw of the Mann-Whitney test for two distributions

that are obviously different, but have relatively similar ranks. Again, in general, the assumption that the two distributions being compared will have identical shape and scaling is unlikely to be satisfied, meaning the Mann-Whitney test is a poor choice for the comparison of the generative spaces. (Notwithstanding earlier work of mine that utilized this test [194]). The failings of the Mann-Whitney test also hold true for the Kruskal–Wallis one-way analysis of variance [106], an extension of the Mann-Whitney test that allows for the comparison of multiple groups at once, as it is based on the same ranking and summing as the Mann-Whitney test.

Given the density estimates via KDE, it would seem desirable to compare them directly. One such method is that of the Wasserstein distance [50] (sometimes referred to as the Earth Movers Distance). The Wasserstein distance can intuitively be thought of as the minimal amount of effort required to transport density from one probability density function so as to make it equivalent to another probability density function (the earth moving metaphor being that it is similar to the amount of effort needed to make one pile of earth look like another). However, for the Wasserstein distance to hold, the density space must have a valid distance measure. For some distributions this makes sense (e.g., in many spaces the Euclidean distance is valid), but this does not hold true for the expressive ranges considered here. What does it mean to move 0.5 units in linearity space and 0.5 units in leniency space?

e-distance by Székely and Rizzo [205] is a statistical comparison meant to alleviate these concerns. e-distance (e for energy, as the metric takes inspiration from Newton's gravitation potential energy) is rotationally invariant in a multidimensional

space, meaning that it is not subject to the distance scaling concerns of the Wasserstein distance. For two d-dimensional random variables, X, Y, the e-distance is defined as:

$$\mathcal{E}(X,Y) = E|X-Y|_d - E|X-X'|_d - E|Y-Y'|_d$$

Where X', Y' are independent identically distributed copies of X, Y, respectively.

Using this notion of e-distance, a non-parametric statistical test for the equivalence of multi-variate distributions can be constructed as:

$$e(X,Y) = \frac{2}{n_i n_j}\sum_{i=1}^{n_1}\sum_{j=1}^{n_2}|X_i-Y_j| - \frac{1}{n_1^2}\sum_{i=1}^{n_1}\sum_{j=1}^{n_1}|X_i-X_j| - \frac{1}{n_2^2}\sum_{i=1}^{n_2}\sum_{j=1}^{n_2}|Y_i-Y_j|$$

which produces a T statistic

$$T_{n_1 n_2} = \frac{n_1 n_2 e(X,Y)}{n_1+n_2}$$

and the null hypothesis of equal distributions is rejected for large $T_{n_1 n_2}$. p-values are given via a bootstrap [58] procedure where the populations are repeatedly resampled B times, the $T_{n_1 n_2, 0<b<B}$ is calculated for each resampling, and the hypothesis is rejected if the observed $T_{n_1 n_2}$ is greater than $100(1-\alpha)\%$ of the resampled values (for confidence level α).

This statistic is exactly what is desired for the comparison of two generative spaces.

- It makes no assumptions about the shape or scale of the distributions
- It allows for a multi-dimensional comparison of metric spaces
- It is rotation invariant and scale equivariant
- It provides a distance metric in addition to a statistical test, allowing for the

comparison of how close a generative space is to another even if a statistically significant difference is found.

Table 11.1 shows the results of an experiment comparing multiple generators' multivariate expressive range distributions against that of the original rooms from *Super Mario Bros.*. The experiment compares the distributions using e-distance for a linearity, leniency, number of jumps, and negative space. Lower e-distance represents a smaller distance between the distributions, leading to a higher probability that null hypothesis of equal distributions is not rejected. Included for comparison is a 10-fold bootstrapping whereby the original levels are repeatedly split into different groups and compared against each other – shown is the average e-distance and associated p-value.

Included in this experiment are two *Kamek* generators – the 512 Unit, 80% dropout, *Snaking*, *Depth*, *Path* and the 64 Unit, no dropout, *Snaking*, *Depth*, *Path*. The 512 is included as it was the generator of highest quality found in the early experiments shown in section 6.3, and the 64 unit is included as a baseline, since it was previously perceived to be of worse quality, it should be expected to have worse results than that of the 512 unit *Kamek* generator. Also included are two MdMC generators by Snodgrass and Ontañón – a baseline MdMC generator discussed in [184] and a MdMC generator with playability constraints as discussed in [185] – and the generator of Riedl and Guzdial [75].

We see that both of the two MdMC methods are found to be statistically different from the original rooms, as is the 64 unit *Kamek* generator, and the generator

of Guzdial and Riedl. However, we see that the 512 unit *Kamek* generator is not found to be statistically significantly different from the original levels. In fact, it is closer to the distribution of original rooms than a repeated sampling of the original rooms against themselves. This is in part due to the wide variability in the original rooms and, certainly, some subsets of the original rooms are much closer (e.g., *e*-distance 3.5136, *p*-value 0.9302) than others (e.g., *e*-distance 18.015, *p*-value 0.04651).

Data Set	*e*-distance	*p*-value
Kamek 512 Unit, 80% dropout, *Snaking*, *Depth*, *Path*	8.2941	**0.4419**
Kamek 64 Unit, no dropout, *Snaking*, *Depth*, *Path*	366.21	0.0233
MdMC 14 Row Splits, Look Ahead 3	560.85	0.0233
MdMC 14 Row Splits, Look Ahead 3, Playability Constrained	44.592	0.04651
Guzdial	497.15	0.0233
Original Levels 10-fold Bootstrapping	12.039	**0.1395**

Table 11.1: Comparison using *e*-distance for a linearity, leniency, number of jumps, and negative space. Lower *e*-distance represents a smaller distance between the distributions, leading to a higher probability that null hypothesis of equal distributions is not rejected.

As discussed in section 11.1, visual inspection of the expressive range of a generator is a useful practice that, while unable to provide a rigorous statistically sound comparison, can help supplement a test such as the one just performed. Figure 11.9 shows the expressive range for the the original rooms, and the two best performing generators from the experiment, the Playability Constrained MdMC, and 512 unit *Kamek* 512 unit generator and the original rooms. Note that this visual inspection is only of two of the dimensions covered in the multidimensional equal distribution test, linearity and leniency. However, even with this subset, we see that the MdMC generator, while sampling heavily in the region of highest density from the original rooms has poor support for the full expressive range found in the originals. The *Kamek* generator, on

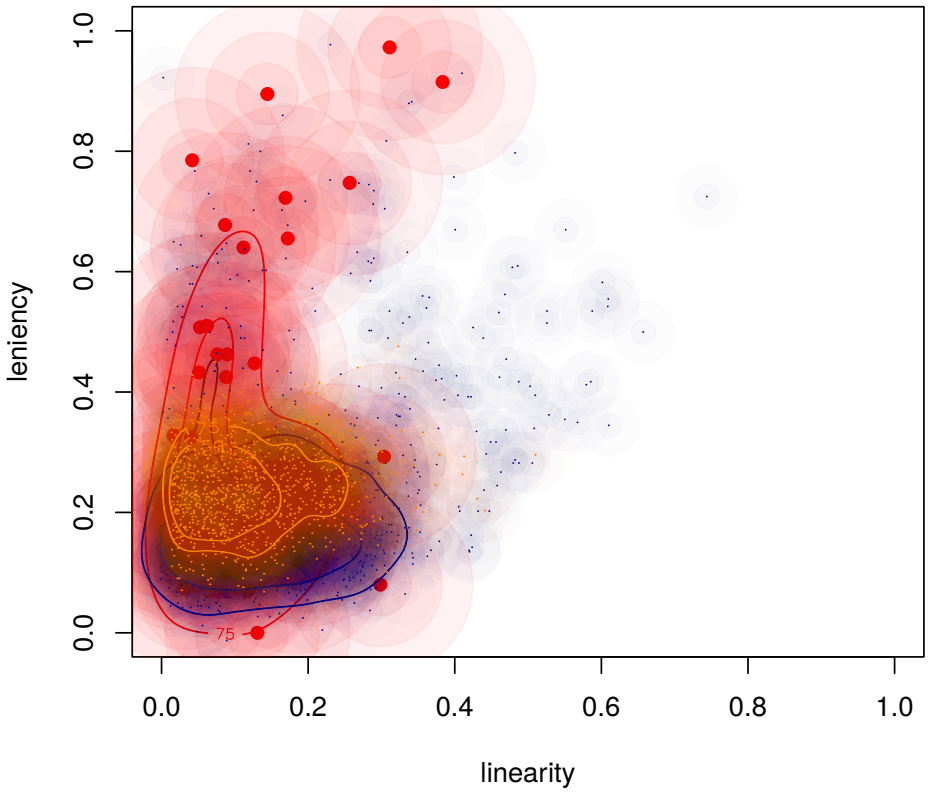


Figure 11.9: Visual inspection of the expressive range of the original rooms from *Super Mario Bros.* (Red), the Playability Constrained MdMC (Orange), and the *Kamek* 512 unit generator (Blue).

.

the other hand, while not sampling as densely in the highly lenient region, nonetheless has closer support to that found by the originals. It is certainly a victory for *Kamek* to produce expressive distributions that are statistically indistinguishable from those of the training corpus. However, while the analysis and understanding of the expressive range of a generator is necessary for the understanding of a generator, it is insufficient. It is still necessary to examine the generated content itself. In the following chapter, I will discuss methodologies for choosing content in a way that helps to bolster the understanding of the capabilities of a generator.

Chapter 12

Analysis of Individual Pieces of Content

The previous chapter discussed methods for the large-scale analysis and comparison of the generative space covered by a generator. While this is certainly a useful practice for PCG researchers and practitioners, at some level, all that matters is the actual individual pieces of content themselves. However, the practice of showing content for a generator is a fraught one, as so far there has been little principle in actually assessing the quality of a generator via the individual content.

Typically, researchers will present a handful of pieces of content, of unknown provenance. Where did they come from? Are they representative of the content that the generator actually produces? In the following section, I will discuss methods for avoiding cherry-picking in the presentation of content, in the hopes of advancing the research methodology of the field.

12.1 Avoiding Cherry-Picking

At its core, the problem of avoiding cherry-picking is one of trustworthiness. While we all hope that everyone is acting in a trustworthy manner, it is certainly possible for this trust to be abused. The simplest, and most likely best, manner for combating the problem of cherry-picking content to be shown in a paper, is to make the generator available. The nascent practice of artifact evaluation is becoming more standard [2], whereby researchers open the actual artifact to examination, not just the research paper describing the artifact. For instance, the Artificial Intelligence for Interactive and Digital Entertainment Conference says of artifact evaluation:

> The purposes of the artifact evaluation are:
> - To promote reproducibility of our research results by reviewing the claims made in the paper and how well they are supported by the corresponding software;
> - To promote reuse by encouraging authors to release software that is well-documented and easy to use by peers;
> - To recognize software artifacts as important scholarly contributions in addition to the paper itself.
>
> [2]

of which the first is the foremost reason considered here: to help review the claims made in the paper, in a way that is not possible with a small set of at worst cherry-picked pieces of content.

Given this, in an ideal situation researchers would make their generators available for others to use, allowing others to explore the generative space of the generator, one piece of content at a time. Unfortunately, this is a difficult ask, in practice. Researchers might not wish to make their code available, perhaps due to its non-production

ready nature, a desire to keep some sort of "competitive advantage," or a legal prohibition on making the code public.

While the act of making the software or code available might be beyond the scope, some have proposed content generation as a service [110, 173] – which, while not making the code publicly available, allows users to explore the generative space, one generation act at a time. However, while this is a wonderful idea, the additional time and monetary demands of turning a generator into an on demand service and hosting it might not be achievable by all researchers.

For researchers unwilling or unable to make their generators available, the option still remains to make a large range of generated content available. Most generators are able to generate content at a very reasonable rate (somewhere between seconds and minutes per piece of content) – allowing for a "large" (large in the sense of what a human would care to sift through, not large in the scale of enterprise data storage) number of pieces to be available for the perusal of others.

While readers are free to peruse the content to assess whether the featured content is representative, it puts the onus on them to dig through a large amount of content. In other fields, the registration of studies is a nascent method for removing self-censorship and observation bias [113]. In these fields, one registers one's study design and hypotheses ahead of time, and then all data and conclusions are made available once the study has been performed. The goal is to reduce practices such as p-hacking [79] wherein studies are run without a firm hypothesis and then the data is manipulated to find statistically significant effects (usually not using multiple testing corrections such

as the Bonferroni correction [28]).

Similarly, it might be the case that PCG researchers could make a set of content available, and then state that the shown content will be based on some predefined random seeding – like SHA-256 [76] hashing on the name of the paper – and then showing some not insignificant number of random pieces of content. However, while the manipulation of this is unlikely in practice, it is possible that researchers could reorder their presented content, so as to still cherry-pick the best content.

The PCGML community as a whole has not adopted any methodology towards addressing these concerns and up to now have taken an ad-hoc approach toward choosing content. Snodgrass and Ontañón [180, 184, 181] show content based on parameterizations of their models. Similarly, Spitaels [187] also showed example rooms based on model parameterization. Hoover et al. chose a famous piece of content to build off of [84]. Guzdial and Riedl choose "representative" content [71, 75], but make no distinction as to what that means. While Lee et al. [200] are not generating content whole cloth – rather adding on to existing content – they choose to show the best and worst results of their method (but since their goal is prediction, they have a clear methodology for best and worst).

Given this, the question still remains, "How can content be selected for inclusion in a research document, such as to allay fears of cherry picking?" While the approach of Lee et al. seems like a good one, for open-ended content generation, it is hard to know what is the "best" and what is the "worst" content to select. The following sections will discuss two ways of selecting from a pool of content, such that the

worst possible selections are presented (for various definitions of worst). These methods provide a lower-bound on the capabilities of a generator.

12.2 Plagiarism As Selection Criterion

For machine learned PCG systems plagiarism (or memorization) is a very real problem. As discussed in the earlier chapters, a major goal for PCGML systems is to generalize from a – usually small – pool of input, so as to be able to generate new content that is similar, but not identical (i.e., it could conceivably have come from the same designer). While it is undesirable for a PCGML system to copy wholesale from the input corpus, it is something of a philosophical question as to what the dividing line is that delineates between a piece of content that is copying from the input corpus and that which has learned from and generalized.

The worst possible case of plagiarism would be the *Copy* generator discussed in chapter 5.2. This generator keeps a copy of the input corpus and selects a member of the corpus and produces it as content. It is trivial to declare this as plagiarism. Alternatively, consider this dissertation. Certainly, all of the individual words found in this document can be found in other documents. The same is most likely true for a large percentage of the pairs of words. So on for triplets of words. While these base atoms of content (words) are found in other documents, it would be absurd to claim this as plagiarism. That being said, there is certainly some point at which a certain string of words being found together would be plagiarism. 10? Maybe, if it were a very

important 10 words. 100? It would have to be a very extenuating set of circumstances for this to not read as plagiarism. 1000? Most certainly.

While an interesting philosophical question, the exact specifics of the dividing line between what is or is not plagiarism do not matter too greatly here, as the goal is not to make a test for whether a piece of content has been plagiarized or not. Instead, the idea is to present the piece of content that is most plagiarized (along with its source of plagiarism), and instead allow the reader to determine whether a piece has been plagiarized or not.

For a PCGML system, this process is relatively straight-forward. The author selects an original piece of content, according to some criterion, and then shows it alongside the generated content that has the most amount of plagiarism from that piece of content.

Potential criteria for selecting the exemplar content:

- The pair that has the most plagiarism – For each piece of content in the input corpus, find the piece of generated content with the most plagiarism. Present the pair with the highest amount of plagiarism (absolute or as percentage of the content pieces – for variable sized content)
- Pick a particularly important piece of original content – Perhaps there is a particularly iconic piece of content (e.g., *Super Mario Bros.* World 1-1). Find the piece of generated content that has the highest amount of plagiarism from the iconic content.

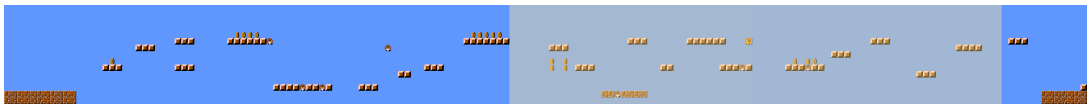
- The most self-plagiarized original content – For all pairs of content from the input corpus, find the pair that plagiarize the most from each other. Find the piece of generated content that has the highest amount of plagiarism from one of (or both) of the pieces of highly self-plagiarized content.

(a) World 2-3 from *Super Mario Bros.*

(b) World 7-3 from *Super Mario Bros.*

Figure 12.1: The two rooms from *Super Mario Bros.* with the highest amount of plagiarism. While, at first glance, they appear identical, the lower room has additional enemies.

(a) Random seed 692 from *Kamek* 512.



(b) World 1-3 from *Super Mario Bros. 2 (JP)*.

Figure 12.2: The room generated by the *Kamek* 512 generator with the highest amount of plagiarism from the original rooms (top) and the room it takes from (bottom). While the portion is of decent size (45% of the original room), the generated room is very different, using the copied piece as a minor segment of the total room.

This last criterion gives us insight into providing a guiding principle for when plagiarism is an issue. By assessing the amount of plagiarism found in the original dataset, we can determine how much plagiarism might be expected from a similar piece of content. While, in a vacuum, it might seem bad for a room to copy 25% of its geometry from a piece of the original content, if the average amount of self-plagiarism

(a) Random seed 100 from *Kamek* 512.

(b) Random seed 478 from *Kamek* 512.

Figure 12.3: The two rooms generated by the *Kamek* 512 generator with the highest amount of self plagiarism. The two rooms start off identically (the first 20% of the rooms are identical), but they quickly diverge. This amount of plagiarism is not too terribly different than that found between the original rooms.

is 35%, then this is to be expected. Figure 12.1 shows the two rooms from *Super Mario Bros.* with the highest amount of self-plagiarism (570 tiles in total, or 16% of the level). By presenting the piece of generated content that takes the most amount of content, a reader can inspect the content to determine the degree to which the generator is memorizing from the original dataset. Figure 12.2 shows the room generated by the 512 unit *Kamek* generator and the room from the original *Super Mario Bros.* with the highest amount of plagiarism.

Of course, this detection of self-plagiarism is not limited to the original dataset, and another possible method for presenting content is to present the two pieces of generated content that are the most highly self-plagiarized. Where comparing the generated content to the originals helps assess the degree of memorization, comparing within a generated set helps assess the degree of mode collapse occurring within the generator. Consider the hypothetical *Super Mario Bros.* generator that predicts the most common room – all sky and ground. The amount of plagiarism in this generated content is unlikely to be particularly high, but all of the generated content will be 100% self-plagiarized. By showing the content that contains the largest amount of copying within

a generated set, readers can come to understand the support and generality of the generator. Figure 12.3 shows the rooms generated by the 512 unit *Kamek* generator with the highest amount of self plagiarism.

There are many possible ways to assess plagiarism, but a general method suitable for two-dimensional grid based room (of the sort that *Mappy* and *Kamek* are designed to work with) is as follows:

$s \leftarrow (1, 1)$
$b \leftarrow \emptyset$
for $r_1, r_2 \in R_1 \times R_2$ **do**
    $s_0 \leftarrow s$
    $l \leftarrow$ all regions of dimension s in r_1 that have are found in r_2
    while $|l| > 0$ **do**
        $s_p \leftarrow s$
        expand s along the smaller dimension, unless that expands it beyond the bounds of either r_1, r_2
        $l \leftarrow$ all regions of dimension s in r_1 that have are found in r_2
    end while
    $s \leftarrow s_p$
    if $s \neq s_0$ **then** $b \leftarrow (r_1, r_2)$
    end if
end for

Simply, for all rooms, r_1, r_2, in the two considered sets of rooms, R_1, R_2, expand a window s until no more overlaps are found. If s grew during the consideration of r_1, r_2, then a new pair of rooms have been found with more overlap than the previous pair with the most overlap.

While determining the amount of plagiarism between a generator and its input corpus is important – as is determining the amount of repetition in the generative space – it is a measure that tends to focus on small sub-pieces of content (as seen in figures 12.2 and 12.3). But there exist other methodologies for the selection and presentation

of content – given the focus on metrics for large scale analysis (both qualitatively and quantitatively), tying them into the individual pieces of content is important.

12.3 Metric Distance as Selection Criterion

Given the importance placed on metrics as evaluative criteria for understanding the generative space of a generator, it seems important to map this understanding back to the individual pieces of content. While acting as a principled methodology for selecting which content to showcase, it also acts as a check on the metrics themselves.

The *Exemplar* methodology is quite simply:

1. Choose an exemplar piece of content – say a particularly important piece of content from the input corpus
2. Using the metrics used elsewhere in the analysis of the generator, find the generated content that is closest to that piece of content or furthest to that piece of content

The *Closest* methodology, to find the piece of content most like any of the original content:

1. For all pieces of original content find the generated content that is closest to that piece of content

Or the *Furthest* methodolgy to find the generated content furthest from the all of the pieces of original content:

1. For all pieces of generated content find the original content with which it is closest.

2. Find the generated content that is further from its closest content than any other piece of generated content

These three methods each answer different questions about the generative space of the generator.

- *Exemplar* – The most subjective of the methodologies, but it can help to showcase particular strengths and weaknesses of the generator by choosing exemplars that have certain unique properties (e.g. choosing a room in the original corpus that is singular and showing how the generator either did or did not learn to match the properties of that content.)

- *Closest* – This acts as both a check on plagiarism (although at a more global scale) and on the metrics themselves. If two pieces are found to be very similar in metric space but do not seem so in visual inspection, then, obviously, there is some aspect of the metrics that is poorly related to the actual qualities that are being attempted to be measured.

- *Furthest* – This acts as both a check on the generalization of the generator and as a sanity check on the generator. If the furthest piece of generated content still appears very similar to all of the original content, then the generator is not learning to generalize beyond the bounds of the original content. On the other hand, if the furthest piece of content is incomprehensible, then the generator is not doing a sufficient job of learning the latent design

The distance calculation between the metrics for one piece of content, m_a, and another, m_b, is

$$V = var(M)$$

$$d = ||(m_a - m_b)V^{-1}||_2$$

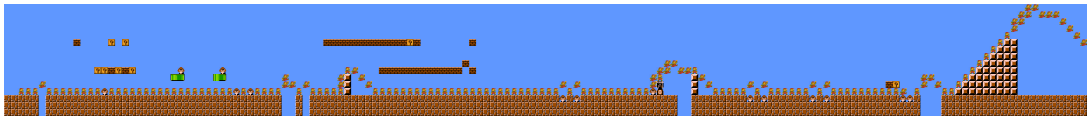
Where M is the set of metrics for all pieces of content. This is the Euclidean distance ($||\quad||_2$, or L2 norm) of the whitened metric vectors. The whitening process (multiplying the difference by the inverse of the covariance matrix) means that all of the metric dimensions will be of the same scale (having variance one), meaning that the original scaling of metrics is unimportant and the distances will not be affected by different scales (some metrics that would normally be positive, like number of jumps could now be negative). Whitening is a common practice in machine learning techniques that rely on distance (e.g. k-means [39], k nearest neighbors [68], etc.)

The following section will use these criteria to show content from *Kamek* and the related pieces of original content.

12.3.1 Case Study – *Kamek*

The single most famous piece of content in the context of platformer games, is that of *Super Mario Bros.*'s World 1-1 [22, 227]. The iconic first set of question mark blocks, bricks, goomba, and mushroom act as gradual introduction for the player to the

(a) World 1-1 from *Super Mario Bros.*



(b) Random seed 447 from *Kamek* 512.

Figure 12.4: World 1-1 and the *Kamek* 512 generated room closest to it in the metrics of negative space, linearity, leniency, and jumps. The generated room has a few more gaps but is a relatively straight forward experience, like 1-1.

world of *Super Mario Bros.* Given its iconic nature, it makes sense to include it as an exemplar piece of content for comparison. Figure 12.4 shows 1-1 and the closest piece of content generated by *Kamek* 512 along the metrics of linearity, leniency, negative space, and number of jumps, with a distance of 0.28. Many of the same structures exist in both and the generated room has a small amount of jumps and enemies, as in 1-1. For the whitened metric vector of:

$$< j, n, R^2, l >$$

World 1-1 has a metric vector of:

$$< -0.97, 1.0, -0.40, -0.97 >$$

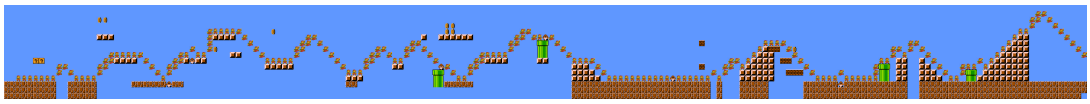
and the closest *Kamek* 512 room has a metric vector of:

$$< 0.81, 0.82, -0.36, -0.84 >$$

leading to a distance of 0.28. The generated room is less extreme in all of the metrics, except linearity, where it is slightly more linear. But it is less lenient, has less negative space, and has fewer jumps than in World 1-1.

While picking a famous piece of content can establish a good baseline, it can

(a) World 5-3 from *Super Mario Bros.*



(b) Random seed 901 from *Kamek* 512.

Figure 12.5: World 5-3 and the *Kamek* 512 generated room closest to it in the metrics of negative space, linearity, leniency, and jumps. The generated room is not solely like 5-3 (it has segments with ground) but is generally very similar to the mostly sky and floating platform filled 5-3.

also be important to show the breadth of content that a generator is capable of creating. For instance, *Super Mario Bros.* has a set of different room motifs (e.g., "normal," underwater, below ground, tree top, dungeon, etc.), and it is important to check to see if a generator is capable of achieving similar styles or content. Figure 12.5 shows an example of a "treetop" room from *Super Mario Bros* and the closest generated by *Kamek*. The second half of the generated room is very similar to the floating platform filled structure of the original room, but there are definitely more portions of the generated room that have ground than found in the original. In part, this is because the chosen metrics do not directly account for the empty space – negative space looks at what proportion of the empty space is reachable.

The metric vectors for World 1-3 and *Kamek* 512 seed 901 are:

$< -0.43, 1.7, -0.64, 1.6 >$

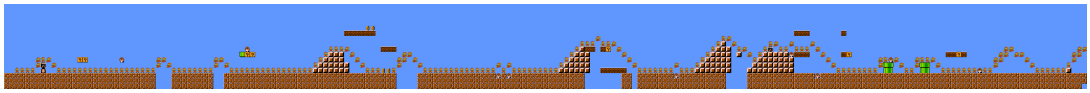
and

$< 0.033, 1.7, -0.43, 1.8 >$

respectively, which is identical for less negative space and very similar linearity and

leniency, but with fewer total jumps, leading to a distance of 0.54.

(a) World 4-2 from *Super Mario Bros. 2* (JP)



(b) Random seed 186 from *Kamek* 512.

Figure 12.6: The room generated by *Kamek* 512 (bottom), closer to any of the original content than any other piece of generated content, and its closest touchstone, World 4-2 (top)in the metrics of negative space, linearity, leniency, and jumps.

Using the *Closest* methodology, we find the generated room that is closer to any of the original content than any other piece of generated content, seen in figure 12.6.

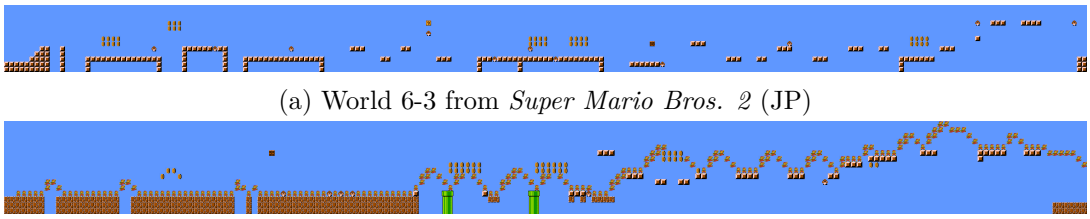
The metric vectors for World 4-2 and *Kamek* 512 seed 186 are:

$< -0.092, -0.11, -0.67, -0.47 >$

and

$< -0.13, -0.22, -0.76, -0.57 >$

respectively. We see that 186 and 4-2 are very close on all of the metrics, with a total of 0.176. The rooms have similar amounts of gaps and enemies, are both heavily ground based, and differ most in the amount of negative space (with world 4-2 having slightly more negative space). Interestingly, the rooms are not particularly similar visually, which can be seen as a merit for the metrics (two seemingly disparate rooms are nonetheless similar in their play) or as distinct failing (the metrics fail to distinguish between two rooms that should obviously be far apart). Again, it remains future work in determining valuable metrics to disentangle this problem.



(a) World 6-3 from *Super Mario Bros. 2* (JP)

(b) Random seed 177 from *Kamek* 512.

Figure 12.7: The room generated by *Kamek* 512 (bottom), further away from the original content than any other, and its closest touchstone, World 6-3 (top)in the metrics of negative space, linearity, leniency, and jumps. The generated room has large gaps and progresses from a relatively straight forward ground based room to a floating platform based room at the final third, as in World 6-3.

Using the *Furthest* methodology, we find the generated room that is further away from any of the original content than any other piece of generated content. The piece of *Kamek* 512 content whose closest touchstone in the original corpus is further than all other generated contents', and its closest relative can be seen in figure 12.7.

The metric vectors for World 6-3 and *Kamek* 512 seed 177 are:

$< 0.37, 0.29, 1.9, 4.5 >$

and

$< -0.47, -0.15, 4.2, 1.8 >$

respectively, and we can see in part why seed 177 is so far away from all of the originals, as it is extremely more linear and much more lenient than most other rooms. The closest room to it is much less linear, while still being more linear than most other rooms, while having much higher leniency than most other rooms. The total distance between the rooms is 3.6, which is much, much higher than any of the discussed differences.

Of course, the choice of metrics – and even weighting of the metrics – is at the discretion of the authors. As discussed in chapter 10, it remains very much an

open question as to what the best metrics are for analyzing generated content, and I would hope that this methodology would be incorporated in the discussion of new metrics. When constructing a new metric, it would be beneficial if authors showed how this new metric is reified in the content, allowing readers to develop intuition, in addition to more rigorous quantitative analyses. To provide an example for metrics that are marginally well understood, consider the example of the *Furthest* rooms shown in figure 12.7. Both the generated room and the exemplar room are outliers in terms of linearity and leniency – by showing these examples, a user begins to understand how these metrics are reified in the extreme – long, sloping rooms that are unlike any other rooms found in the dataset (normally filled with flat stretches that have sporadic bursts of complexity). In developing new metrics, showing this type of extreme output helps a user gain intuition into what qualia the metric is trying to capture.

12.4 Discussion

This last chapter and chapter 11.3 present a suite of techniques for evaluating and analyzing the capabilities of procedural content generators, with a specific focus on techniques for machine learning based generators. A key takeaway from these chapters is that no one technique suffices for the evaluation of PCG(ML) system. Viewing individual pieces of content, no matter how meaningfully chosen, provides a minuscule keyhole to attempt to begin the full range of a generative system. Large scale expressive range style analyses provide a high level overview, but misses the trees from the forest,

providing insight into the shape of a generative space but not into individual pieces of content. A successful analysis of a PCGML system should:

ML1 Address the extent of memorization of the system

ML2 Assess the generator's ability to generalize

ML3 Provide an overview of how well the generator has learned the latent design

Similarly, a successful analysis of a classical PCG system should:

C1 Address the repetition of the system

C2 Assess the generator's ability to generate novel content

C3 Provide an overview of the generator's style

With clear analogues between **ML1** and **C1**, **ML2** and **C2**, and **ML3** and **C3**. For PCGML systems, **ML1** is best addressed by showing the piece of content with the most plagiarism, as that is how the memorization is reified in the generated artifact. **ML2** is best addressed by showing the piece of content furthest from any of the original content, as that shows how well the generator is able to generate content outside of the bounds of the training corpus. While **ML3** does not require both qualitative and quantitative analyses, it is best served by both a visualization of the generative space and the training space – hopefully, in a number of meaningful metrics – via a corner plot and a quantitative measure via the e-distance.

For classical PCG systems, **C1** is best addressed by showing the pieces of content with the most self-plagiarism, allowing users to assess how much the system

repeats itself. **C2** is best addressed in conjunction with **C3** – **C3** should be visualized via a corner plot of the generative spaces – and locations in that generative space should be chosen as exemplars via the metric distance methodology discussed in the last chapter, showing the range of generatable content. It should be noted that the only major changes here compared to the work of Smith and Whitehead is the extension in to higher dimensionality and the assessment of self-plagiarism.

Chapter 13

Conclusions And Future Work

This dissertation has aimed to address the broad question of:

Can a machine learn to generate game map content from observation without human intervention?

Now, obviously, this is a lofty goal not entirely answered by the work presented here. However, this dissertation marks a number of advances towards achieving this goal:

- – A contribution to game studies, a theoretical framework for the phenomenology for how humans perceive and understand game maps (Chapter 2)
- A contribution to the field of Artificial Intelligence, presenting a proceduralization of said processes in the first system that can take in a game and input trace and produce human understandable annotations of the content encountered, *Mappy* (Chapter 3)
- A contribution to the field of procedural content, a theoretical understanding

of what is required for a machine learned two-dimensional room generator (the most common class of PCGML system found) to achieve generality, i.e., have no theoretical limitations on its generative capabilities (Chapter 5)

- A contribution to the field of procedural content genreation, a general two-dimensional room generator, *Kamek*, that is capable of generating high quality content, along with a thorough experimentation of how different parameter and data representation choices affect the abilities of the generator (Chapter 6)

- A contribution to the field of procedural content generation, a theoretical understanding of what is required for a machine learned map (the graphical structure composed of rooms and linkages) generator to achieve generality (Chapter 7)

- A contribution to the field of procedural content generation, the first machine learned map generator, *Agahnim*, that is capable of producing maps in the style of dungeons from *The Legend of Zelda* (Chapter 8)

- A contribution to the field of procedural content generation, the development of methodologies for understanding the generative properties of a generator. These methodologies encompass understanding the expressive range of a generator in a more concrete and quantitative way than has previously been possible. In particular, the introduction of machinery for determining how close a PCGML system is to the input corpus – i.e., a measure for how well a PCGML system has achieved its task of learning the design latent within the input corpus (Chapter 11).

- A contribution to the field of procedural content generation, methodologies for the display of individual pieces of content, designed to be robust to cherry-picking, that have properties that better enable people to understand the generators that produced that content. (Chapter 12)

These contributions provide theoretical and design contributions that further the understanding of games and procedural content generation therein – along with systems that reify these understandings.

13.1 Future Work

Of course, these contributions do not represent the end point for this field of study, and in fact point to new, exciting potential work. On the topic of game observation and understanding, there are two particularly important avenues: **(1)** automatic experimentation and play and **(2)** understanding of semantics. As mentioned in Chapter 3, general video game playing is a field of study typically devoted towards playing a heretofore unseen game well. While the playing of a game well is a worthwhile topic of research, *Mappy* points towards other ways this research can be utilized, e.g. playing a game to learn as much about it as possible. This can allow for computer and human alike to better understand games, enabling further capabilities for automatic systems like *Mappy*. *Mappy*, like all GVG-AI players, understands games in a syntactic sense – they understand the structure, but not the meaning. This meaning comes from knowledge of the world and culture, two things AI has generally struggled at understanding.

However, the potential gains from incorporating these pieces are huge, as they can enable a system to understand a game in a way much closer to how a human does.

On the topic of PCGML systems, there is still much work to address how decorative content changes the generative landscape of the different generation systems, *Kamek* included. *Agahnim* represents an initial foray into entire map generation – but lacks generality – and future work should focus less on individual rooms and on maps as a cohesive whole, with generation of maps and the individual rooms found in those maps as one holistic generation process.

Finally, I would like to see the adoption of the evaluation methodologies presented in this work. Just as expressive range visualization was a step forward for the PCG research field, I hope the approaches detailed here will present another step forward. And just as expressive range visualizations have been built upon here, I believe that the techniques in this dissertation will be built upon as our understanding of the PCG field deepens, and we identify gaps in that understanding that we weren't able to previously voice.

Bibliography

[1] *4.9 Blueprints Visual Scripting*, (accessed April 30, 2018). https://docs.unrealengine.com/en-us/Engine/Blueprints.

[2] *AIIDE 2018 Artifact Evaluation*, (accessed April 30, 2018). https://sites.google.com/ncsu.edu/aiide-2018/home/artifact-evaluation?authuser=0.

[3] *Anscombe's quartet*, (accessed April 30, 2018). https://en.wikipedia.org/wiki/Anscombe%27s_quartet.

[4] *Any tips for practicing a frame perfect trick?*, (accessed April 30, 2018). https://www.speedrun.com/Speedrunning/thread/uys35/1#fdo5l.

[5] *Association for Computational Creativity*, (accessed April 30, 2018). http://www.computationalcreativity.net.

[6] *Backroom Boys: Masters of their universe*, (accessed April 30, 2018). https://www.theguardian.com/books/2003/oct/18/features.weekend.

[7] *Border Effects*, (accessed April 30, 2018). https://www.mathworks.com/help/wavelet/ug/dealing-with-border-distortion.html.

[8] *Fsm:This class defines a Finite State Machine (FSM)*, (accessed April 30, 2018). https://hutonggames.fogbugz.com/default.asp?W1098.

[9] *GVGAI:The General Video Game AI Competition*, (accessed April 30, 2018). http://www.gvgai.net/.

[10] *IAN-ALBERT.COM*, (accessed April 30, 2018). http://ian-albert.com/games/.

[11] *Logo Programming*, (accessed April 30, 2018). http://el.media.mit.edu/logo-foundation/what_is_logo/logo_programming.html.

[12] *Nes Dev: Mirroring*, (accessed April 30, 2018). https://wiki.nesdev.com/w/index.php/Mirroring.

[13] *scipy.ndimage.filters.convolve*, (accessed April 30, 2018). https://docs.scipy.org/doc/scipy-0.16.1/reference/generated/scipy.ndimage.filters.convolve.html.

[14] *TASVideos:Tool-assisted game movies*, (accessed April 30, 2018). http://tasvideos.org/.

[15] *Unity Documentation Camera*, (accessed April 30, 2018). https://docs.unity3d.com/Manual/class-Camera.html.

[16] *VGMAPS.com: THE VIDEO GAME ATLAS*, (accessed April 30, 2018). http://vgmaps.com/.

[17] Metanet Software. N++, 2015.

[18] Espen J Aarseth. *Cybertext: Perspectives on ergodic literature*. JHU Press, 1997.

[19] Jurandy Almeida, Rodrigo Minetto, Tiago A Almeida, Ricardo da S Torres, and Neucimar J Leite. Robust estimation of camera motion using optical flow models. In *International Symposium on Visual Computing*, pages 435–446. Springer, 2009.

[20] Rajeev Alur, Thao Dang, Joel Esposito, Yerang Hur, Franjo Ivancic, Vijay Kumar, P Mishra, GJ Pappas, and Oleg Sokolsky. Hierarchical modeling and analysis of embedded systems. *Proceedings of the IEEE*, 2003.

[21] FJ Anscombe. Graphs in statistical analysis. *The American Statistician*, 27(1):17–21, 1973.

[22] Anna Anthropy. *level design lesson: to the right, hold on tight*, (accessed April 30, 2018). `https://web.archive.org/web/20090911102829/http://www.auntiepixelante.com/?p=465`.

[23] Sanjeev Arora and Yi Zhang. Do gans actually learn the distribution? an empirical study. *arXiv preprint arXiv:1706.08224*, 2017.

[24] Yaakov Bar-Shalom, X Rong Li, and Thiagalingam Kirubarajan. *Estimation with applications to tracking and navigation: theory algorithms and software*. John Wiley & Sons, 2004.

[25] Shane Barratt and Rishi Sharma. A note on the inception score. *arXiv preprint arXiv:1801.01973*, 2018.

[26] M. Bauerly and Y. Liu. Computational modeling and experimental investigation of effects of compositional elements on interface and design aesthetics. *International Journal of Human-Computer Studies*, 64(8):670–682, 2006.

[27] Richard Bellman and Robert Roth. Curve fitting by segmented straight lines. *Journal of the American Statistical Association*, 1969.

[28] C. E. Bonferroni. Teoria statistica delle classi e calcolo delle probabilità. *Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commerciali di Firenze*, 8:3–62, 1936.

[29] Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. *preprint arXiv:1206.6392*, 2012.

[30] David Braben and Ian Bell. Elite, 1984.

[31] Lisa Gottesfeld Brown. A survey of image registration techniques. *ACM Comput. Surv.*, 24(4):325–376, December 1992.

[32] Eric Butler, Erik Andersen, Adam M Smith, Sumit Gulwani, and Zoran Popović. Automatic game progression design through analysis of solution features. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 2407–2416. ACM, 2015.

[33] Edward Byrne. *Game level design*, volume 6.

[34] Alessandro Canossa and Gillian Smith. Towards a procedural evaluation technique: Metrics for level design. *Proceedings of FDG*, 2015.

[35] José E Chacón and T Duong. Multivariate plug-in bandwidth selection with unconstrained pilot bandwidth matrices. *Test*, 19(2):375–398, 2010.

[36] Nathanael Chambers and Dan Jurafsky. Unsupervised learning of narrative event chains. *Proceedings of ACL-08: HLT*, pages 789–797, 2008.

[37] Nathanael Chambers and Dan Jurafsky. Unsupervised learning of narrative schemas and their participants. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2-Volume 2*, pages 602–610. Association for Computational Linguistics, 2009.

[38] W Ching, S Zhang, and M Ng. On multi-dimensional markov chain models. *Pacific Journal of Optimization*, 3(2), 2007.

[39] Adam Coates and Andrew Y Ng. Learning feature representations with k-means. In *Neural networks: Tricks of the trade*, pages 561–580. Springer, 2012.

[40] Jasmine Collins, Jascha Sohl-Dickstein, and David Sussillo. Capacity and trainability in recurrent neural networks. *arXiv preprint arXiv:1611.09913*, 2016.

[41] Simon Colton. Creativity versus the perception of creativity in computational systems. pages 14–20, 01 2008.

[42] Michael Cook, Jeremy Gow, and Simon Colton. Danesh: Helping bridge the gap between procedural generators and their output. 2016.

[43] WJ Crozier and AH Holway. Theory and measurement of visual mechanisms: Iii. δi as a function of area, intensity, and wave-length, for monocular and binocular stimulation. *The Journal of general physiology*, 23(1):101–141, 1939.

[44] Steve Dahlskog and Julian Togelius. Patterns as objectives for level generation. In *EvoApplications*, 2013.

[45] Steve Dahlskog, Julian Togelius, and Mark J. Nelson. Linear levels through n-grams. In *Proceedings of the 18th International Academic MindTrek Conference*, 2014.

[46] Ernest Davis and Gary Marcus. Commonsense reasoning and commonsense knowledge in artificial intelligence. *Communications of the ACM*, 58(9):92–103, 2015.

[47] Andres Delikat, zeromus, feos, and Lukas Sabota. FCEUX, 2008.

[48] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[49] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[50] Roland L Dobrushin. Prescribing a system of random variables by conditional distributions. *Theory of Probability & Its Applications*, 15(3):458–486, 1970.

[51] Joris Dormans. Adventures in level design: Generating missions and spaces for action adventure games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 1:1–1:8, New York, NY, USA, 2010. ACM.

[52] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[53] Frederic Dufaux and Janusz Konrad. Efficient, robust, and fast global motion estimation for video coding. *IEEE transactions on image processing*, 9(3):497–501, 2000.

[54] Andrew Dunn. *Zoetrope*, (accessed April 30, 2018). `https://commons.wikimedia.org/wiki/File:Zoetrope.jpg`.

[55] Tarn Duong and Martin Hazelton. Plug-in bandwidth matrices for bivariate kernel density estimation. *Journal of Nonparametric Statistics*, 15(1):17–30, 2003.

[56] Tarn Duong and Martin L Hazelton. Convergence rates for unconstrained bandwidth matrix selectors in multivariate kernel density estimation. *Journal of Multivariate Analysis*, 93(2):417–433, 2005.

[57] Jack Edmonds. Maximum matching and a polyhedron with 0, 1-vertices.

[58] Bradley Efron and Robert J Tibshirani. *An introduction to the bootstrap*. CRC press, 1994.

[59] Mustafa Ersen and Sanem Sariel-Talay. Learning interactions among objects through spatio-temporal reasoning. In *Signal Processing and Communications Applications Conference (SIU), 2012 20th*, pages 1–4. IEEE, 2012.

[60] Shai Fine, Yoram Singer, and Naftali Tishby. The hierarchical hidden markov model: Analysis and applications. *Machine learning*, 32(1):41–62, 1998.

[61] Daniel Foreman-Mackey. corner.py: Scatterplot matrices in python. *The Journal of Open Source Software*, 24, 2016.

[62] Goran Frehse. Phaver: Algorithmic verification of hybrid systems past hytech. In *International workshop on hybrid systems: computation and control*. Springer, 2005.

[63] Michael Frey. *Taumatropio fiori e vaso, 1825 Frame 1 & 2*, (accessed April 30, 2018). `https://en.wikipedia.org/wiki/File:Taumatropio_fiori_e_vaso,_1825_Frame_1.png`.

[64] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[65] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[66] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[67] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, and Daan Wierstra. Draw: A recurrent neural network for image generation. *preprint arXiv:1502.04623*, 2015.

[68] Patrick J Grother, Gerald T Candela, and James L Blue. Fast implementations of nearest neighbor classifiers. *Pattern Recognition*, 30(3):459–465, 1997.

[69] Christine Guillemot and Olivier Le Meur. Image inpainting: Overview and recent advances. *IEEE signal processing magazine*, 31(1):127–144, 2014.

[70] Maxim Gumin. WaveFunctionCollapse. *GitHub repository*, 2016. `https://github.com/mxgmn/WaveFunctionCollapse`.

[71] M. Guzdial, N. Sturtevant, and B. Li. Deep static and dynamic level analysis: A study on infinite mario. In *Proceedings of the 3rd Experimental AI in Games Workshop*, page 8, 2016.

[72] Matthew Guzdial, Boyang Li, and Mark O Riedl. Game engine learning from video. In *26th International Joint Conference on Artificial Intelligence*, 2017.

[73] Matthew Guzdial and Mark Riedl. Game level generation from gameplay videos. In *AIIDE*, 2016.

[74] Matthew Guzdial and Mark Riedl. Learning to blend computer game levels. 2016.

[75] Matthew Guzdial and Mark O. Riedl. Toward game level generation from gameplay videos. In *Proceedings of the FDG workshop on Procedural Content Generation in Games*, 2015.

[76] Helena Handschuh. Sha-0, sha-1, sha-2 (secure hash algorithm). In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security (2nd Ed.)*, pages 1190–1193. Springer, 2011.

[77] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2):100–107, 1968.

[78] Daniel M Hausman and James Woodward. Independence, invariance and the causal markov condition. *The British journal for the philosophy of science*, 50(4):521–583, 1999.

[79] Megan L Head, Luke Holman, Rob Lanfear, Andrew T Kahn, and Michael D Jennions. The extent and consequences of p-hacking in science. *PLoS biology*, 13(3):e1002106, 2015.

[80] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, Günter Klambauer, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a nash equilibrium. *arXiv preprint arXiv:1706.08500*, 2017.

[81] William Higinbotham. Tennis for Two. [Analog Computer], 1958.

[82] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computing*, 1997.

[83] Amy K. Hoover, Paul A. Szerlip, and Kenneth O. Stanley. Functional scaffolding

for composing additional musical voices. *Computer Music Journal*, 38(4):80–99, 2014.

[84] Amy K Hoover, Julian Togelius, and Georgios N. Yannakis. Composing video game levels with music metaphors through functional scaffolding. *Comp. Creativity & Games at ICCC*, 2015.

[85] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[86] Britton Horn, Steve Dahlskog, Noor Shaker, Gillian Smith, and Julian Togelius. A comparative evaluation of procedural level generators in the Mario AI framework. *FDG*, 2014.

[87] Ian D. Horswill and Leif Foged. Fast procedural level population with playability constraints. In Mark Riedl and Gita Sukthankar, editors, *AIIDE*. The AAAI Press, 2012.

[88] Xun Huang, Yixuan Li, Omid Poursaeed, John Hopcroft, and Serge Belongie. Stacked generative adversarial networks.

[89] hutong games. playMaker, 1993.

[90] id Software. Doom, 1993.

[91] Itseez. Open source computer vision library. `https://github.com/itseez/opencv`, 2015.

[92] Rishabh Jain, Aaron Isaksen, Christoffer Holmgård, and Julian Togelius. Autoencoders for Level Generation, Repair, and Recognition. In *ICCC*, 2016.

[93] jdaster64. SMB physics spec. http://forums.mfgg.net/viewtopic.php?p=346301, 2012. Accessed: 2017-02-13.

[94] Jeff Tunnell Productions. The Incredible Machine, 1993.

[95] Susmit Jha, Bryan A Brady, and Sanjit A Seshia. Symbolic reachability analysis of lazy linear hybrid automata. In *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer, 2007.

[96] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.

[97] Evangelos Kalogerakis, Siddhartha Chaudhuri, Daphne Koller, and Vladlen Koltun. A probabilistic model for component-based shape synthesis. *ACM Transactions on Graphics*, 31(4), 2014.

[98] Andrej Karpathy, Justin Johnson, and Fei-Fei Li. Visualizing and understanding recurrent networks. *CoRR*, abs/1506.02078, 2015.

[99] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*, 2017.

[100] Leonard Kaufman and Peter Rousseeuw. *Clustering by means of medoids*. North-Holland, 1987.

[101] Itay Keren. *Scroll Back: The Theory and Practice of Cameras in Side-Scrollers*, (accessed April 30, 2018). https://www.gamasutra.com/blogs/ItayKeren/20150511/243083/Scroll_Back_The_Theory_and_Practice_of_Cameras_in_SideScrollers.php#h.nc2c2clexluw.

[102] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[103] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[104] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[105] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg, 2006. Springer-Verlag.

[106] William H Kruskal and W Allen Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, 47(260):583–621, 1952.

[107] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[108] Daniel D Lee and H Sebastian Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–791, 1999.

[109] Richard J Leskosky. Phenakiscope: 19th century science turned to animation. *Film History*, 5(2):176–189, 1993.

[110] Antonios Liapis. Map sketch generation as a service. 2015.

[111] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. Sentient sketchbook: Computer-aided game level authoring. In *FDG*, pages 213–220, 2013.

[112] Roland van der Linden, Ricardo Lopes, and Rafael Bidarra. Designing procedurally generated levels. In *Proceedings of IDPv2 2013 - Workshop on Artificial Intelligence in the Game Design Process, co-located with the Ninth AAAI Conference on Artificial Intelligence in Interactive Digital Entertainment*, pages 41–47, AAAI Press, Palo Alto, CA, oct 2013. AAAI, AAAI Press. ISBN 978-1-57735-635-6.

[113] Elizabeth Loder. Registration of observational studies. *BMJ*, 340(2):c950, 2010.

[114] Kaifeng Lv, Shunhua Jiang, and Jian Li. Learning gradient descent: Better generalization and longer horizons. *arXiv preprint arXiv:1703.03633*, 2017.

[115] Daniel L Ly and Hod Lipson. Learning symbolic representations of hybrid dynamical systems. *Journal of Machine Learning Research*, 13(Dec):3585–3618, 2012.

[116] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[117] Julian RH Mariño and Levi HS Lelis. A computational model based on symmetry for generating visually pleasing maps of platform games. 2016.

[118] Julian RH Mariño, Willian MP Reis, and Levi HS Lelis. An empirical evaluation of evaluation metrics of procedurally generated mario levels. *The AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2015.

[119] Michael Mateas and Noah Wardrip-Fruin. Defining operational logics. 2009.

[120] Justin Matejka and George Fitzmaurice. Same stats, different graphs: generating datasets with varied appearance and identical statistics through simulated annealing. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 1290–1294. ACM, 2017.

[121] Peter Mawhorter and Michael Mateas. Procedural level generation using occupancy-regulated extension. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 351–358. IEEE, 2010.

[122] Thomas McClean. *McClean's Optical Illusions*, 1833.

[123] Gábor Melis, Chris Dyer, and Phil Blunsom. On the state of the art of evaluation in neural language models. *The Sixth International Conference on Learning Representations*, 2018.

[124] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing lstm language models. *arXiv preprint arXiv:1708.02182*, 2017.

[125] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[126] T. Minka, J.M. Winn, J.P. Guiver, and D.A. Knowles. Infer.NET 2.5, 2012. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

[127] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[128] Jessica Montgomery. Machine learning in 2016: what does nips2016 tell us about the field today?, 2016. https://blogs.royalsociety.org/in-verba/2016/12/22/machine-learning-in-2016-what-does-nips2016-tell-us-about-the-field-today/.

[129] Fausto Mourato, Manuel Próspero dos Santos, and Fernando Birra. Automatic level generation for platform videogames using genetic algorithms. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*, page 8. ACM, 2011.

[130] Namco. Periscope. [Arcade] Namco, 1965.

[131] Namco. Galaxian, 1993.

[132] NESDev. Nesdev wiki: Nes reference guide, 2017.

[133] D. C. L. Ngo, A. Samsudin, and R. Abdullah. Aesthetic measures for assessing graphic screens. *J. Inf. Sci. Eng*, 16(1):97–116, 2000.

[134] David Chek Ling Ngo, Lian Seng Teo, and John G Byrne. Modelling interface aesthetics. *Information Sciences*, 152:25–46, 2003.

[135] Oliver Niggemann, Benno Stein, Asmir Vodencarevic, Alexander Maier, and Hans Kleine Büning. Learning behavior models for hybrid timed systems. In *AAAI*, 2012.

[136] Nintendo RD1. Beneath Apple Manor, 1986.

[137] Richard E Nisbett and Timothy D Wilson. Telling more than we can know: Verbal reports on mental processes. *Psychological review*, 84(3):231, 1977.

[138] Joseph Osborn, Adam Summerville, and Michael Mateas. Automatic mapping of nes games with mappy. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, page 78. ACM, 2017.

[139] Joseph C. Osborn. *OL-Catalog*, (accessed April 30, 2018). `https://github.com/JoeOsborn/ol-catalog`.

[140] Joseph C. Osborn. *OL-Catalog/notes/games*, (accessed April 30, 2018). `https://github.com/JoeOsborn/ol-catalog/blob/master/notes/games.md`.

[141] Joseph C Osborn, Adam Summerville, and Michael Mateas. Automated game design learning. In *Computational Intelligence and Games (CIG), 2017 IEEE Conference on*, pages 240–247. IEEE, 2017.

[142] Joseph C. Osborn, Noah Wardrip-Fruin, and Michael Mateas. Refining operational logics. In *FDG*, 2017.

[143] Karl Pearson. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 58:240–242, 1895.

[144] Alison Pease, Daniel Winterstein, and Simon Colton. Evaluating machine creativity. 2001.

[145] Christopher Pedersen, Julian Togelius, and Georgios N Yannakakis. Modeling player experience for content creation. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):54–67, 2010.

[146] Markus Persson. Infinite Mario, 2006.

[147] Paolo Piselli, Mark Claypool, and James Doyle. Relating cognitive models of computer games to user evaluations of entertainment. In Jim Whitehead and R. Michael Young, editors, *FDG*, pages 153–160. ACM, 2009.

[148] André Platzer. Differential dynamic logic for hybrid systems. *Journal of Automated Reasoning*, 2008.

[149] Kari Pulli, Anatoly Baksheev, Kirill Kornyakov, and Victor Eruhimov. Real-time computer vision with opencv. *Commun. ACM*, 55(6):61–69, June 2012.

[150] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[151] Racconish. *Optical illusion disc with man pumping water*, (accessed April

30, 2018). https://commons.wikimedia.org/wiki/File:Optical_illusion_disc_with_man_pumping_water.gif.

[152] W. M. P. Reis, L. H. S. Lelis, and Y. Gal. Human computation for procedural content generation in platform games. In *Conference of Computational Intelligence and Games*, pages 99–106. IEEE, 2015.

[153] Graeme Ritchie. Assessing creativity. Citeseer, 2001.

[154] Jonathan Roberts and Ke Chen. Learning-based procedural content generation. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(1):88–101, 2015.

[155] Murray Rosenblatt. Remarks on some nonparametric estimates of a density function. *The Annals of Mathematical Statistics*, pages 832–837, 1956.

[156] James Ryan, Eric Kaltman, Andrew Max Fisher, Taylor Owen-Milner, Michael Mateas, and Noah Wardrip-Fruin. Gamespace: An explorable visualization of the videogame medium.

[157] S S. Blackman. Multiple target tracking with radar applications. -1, 01 1986.

[158] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in Neural Information Processing Systems*, pages 2234–2242, 2016.

[159] Pedro Santana, Spencer Lane, Eric Timmons, Brian Williams, and Carlos Forster. Learning hybrid models with guarded transitions. 2015.

[160] Santiago Londoño and Olana Missura. Graph grammars for super mario bros levels. In *Proceedings of the Procedural Content Generation Workshop*, 2015.

[161] Stefan Schaal. Is imitation learning the route to humanoid robots? *Trends in cognitive sciences*, 3(6):233–242, 1999.

[162] Richard Scheines. An introduction to causal inference. 1997.

[163] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[164] Gideon Schwarz et al. Estimating the dimension of a model. *The annals of statistics*, 1978.

[165] N. Shaker, M. Nicolau, G. N. Yannakakis, J. Togelius, and M. O'Neill. Evolving levels for super mario bros using grammatical evolution. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 304–311, Sept 2012.

[166] Noor Shaker and Mohamed Abou-Zleikha. Alone we can do so little, together we can do so much: A combinatorial approach for generating game content. In *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2014.

[167] Noor Shaker, Mohammad Shaker, and Julian Togelius. Evolving playable content for cut the rope through a simulation-based approach. In *AIIDE*, 2013.

[168] Noor Shaker, Mohammad Shaker, and Julian Togelius. Ropossum: An authoring tool for designing, optimizing and solving cut the rope levels. 2013.

[169] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.

[170] Noor Shaker, Julian Togelius, Georgios N Yannakakis, Ben Weber, Tomoyuki Shimizu, Tomonori Hashiyama, Nathan Sorenson, Philippe Pasquier, Peter Mawhorter, Glen Takahashi, et al. The 2010 mario ai championship: Level generation track. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(4):332–347, 2011.

[171] Edgar Simo-Serra, Arnau Ramisa, Guillem Alenyà, Carme Torras, and Francesc Moreno-Noguer. Single image 3d human pose estimation from noisy observations. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 2673–2680. IEEE, 2012.

[172] Robert A Singer. Estimating optimal tracking filter performance for manned maneuvering targets. *IEEE Transactions on Aerospace and Electronic Systems*, (4):473–483, 1970.

[173] Adam M. Smith. Answer set programming in proofdoku. 2017.

[174] Adam M Smith and Michael Mateas. Answer set programming for procedural content generation: A design space approach. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):187–200, 2011.

[175] G. Smith, J. Whitehead, and M. Mateas. Tanagra: Reactive planning and con-

straint solving for mixed-initiative level design. *IEEE Transactions on Computational Intelligence and AI in Games*, (99), 2011.

[176] Gillian Smith. *The Seven Deadly Sins of PCG Research*, (accessed April 30, 2018). `http://sokath.com/main/blog/2013/05/23/the-seven-deadly-sins-of-pcg-papers-questionable-claims-edition/`.

[177] Gillian Smith and Jim Whitehead. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, page 4. ACM, 2010.

[178] Gillian Smith, Jim Whitehead, Michael Mateas, Mike Treanor, Jameka March, and Mee Cha. Launchpad: A rhythm-based level generator for 2-d platformers. *IEEE Transactions on computational intelligence and AI in games*, 3(1):1–16, 2011.

[179] Gillian Margaret Smith. *Expressive design tools: Procedural content generation for game designers*. PhD thesis, University of California, Santa Cruz, 2012.

[180] Sam Snodgrass and Santiago Ontañón. A hierarchical approach to generating maps using markov chains. In *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2014.

[181] Sam Snodgrass and Santiago Ontañón. Learning to generate video game maps using Markov models. *TCIAIG*, 2016.

[182] Sam Snodgrass and Santiago Ontañón. Generating maps using markov chains. In *AIIDE 2013 workshop on AI for Game Aesthetics*, pages 25–28. AAAI, 2013.

[183] Sam Snodgrass and Santiago Ontañón. Experiments in map generation using Markov chains. In *FDG*, volume 14, 2014.

[184] Sam Snodgrass and Santiago Ontañón. A hierarchical MdMC approach to 2D video game map generation. *AIIDE*, 2015.

[185] Sam Snodgrass and Santiago Ontañón. Controllable procedural content generation via constrained multi-dimensional Markov chain sampling. In *25th International Joint Conference on Artificial Intelligence*, 2016.

[186] Nathan Sorenson and Philippe Pasquier. Towards a generic framework for automated video game level creation. In *Applications of Evolutionary Computation*, pages 131–140. Springer, 2010.

[187] Jens Spitaels. Marionet: generating realistic game levels through deep. 2017.

[188] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10:99–127, 2002.

[189] Matthew Stephenson and Jochen Renz. Procedural generation of levels for angry birds style physics games. 2016.

[190] Robert A Stine. Model selection using information theory and the mdl principle. *Sociological Methods & Research*, 2004.

[191] Student. The probable error of a mean. *Biometrika*, pages 1–25, 1908.

[192] David Sudnow. *Pilgrim in the microworld*. Heinemann, London, 1983.

[193] Adam Summerville, Morteza Behrooz, Michael Mateas, and Arnav Jhala. What does that ?-block do? learning latent causal affordances from mario play traces. In *Proceedings of the first What's Next for AI in Games Workshop at AAAI*, volume 2017, 2017.

[194] Adam Summerville, Matthew Guzdial, Michael Mateas, and Mark O Riedl. Learning player tailored content from observation: Platformer level generation from video traces using LSTMs. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.

[195] Adam Summerville, Julian RH Mariño, Sam Snodgrass, Santiago Ontañón, and Levi HS Lelis. Understanding Mario: an evaluation of design metrics for platformers. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, page 8. ACM, 2017.

[196] Adam Summerville and Michael Mateas. Sampling hyrule: Sampling probabilistic machine learning for level generation. *AIIDE*, 2015.

[197] Adam Summerville and Michael Mateas. Super Mario as a string: Platformer level generation via LSTMs. *DiGRA/FDG*, 2016.

[198] Adam Summerville and Michael Mateas. Super mario as a string: Platformer

level generation via lstms. In *To Appear In Proceedings of the First International Conference of DiGRA and FDG*, 2016.

[199] Adam Summerville, Joseph Osborn, and Michael Mateas. Charda: Causal hybrid automata recovery via dynamic analysis. *IJCAI*, 2017.

[200] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. Procedural content generation via machine learning (pcgml). *arXiv preprint arXiv:1702.00539*, 2017.

[201] Adam J Summerville, Morteza Behrooz, Michael Mateas, and Arnav Jhala. The learning of Zelda: Data-driven learning of level topology. *PCG Workshop at FDG*, 2015.

[202] Adam James Summerville, Shweta Philip, and Michael Mateas. MCMCTS PCG 4 SMB: Monte Carlo tree search to guide platformer level generation. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[203] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *CoRR*, 2014.

[204] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, et al. Going deeper with convolutions.

[205] Gábor J Székely and Maria L Rizzo. Energy statistics: A class of statistics based on distances. *Journal of statistical planning and inference*, 143(8):1249–1272, 2013.

[206] Game feel. In Steve Swink, editor, *Game Feel*, pages i – iii. Morgan Kaufmann, Boston, 2009.

[207] Taito. Speed Race, 1974.

[208] Edward Teng and Rafael Bidarra. A semantic approach to patch-based procedural generation of urban road networks. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, page 71. ACM, 2017.

[209] Carsten Thomassen. The graph genus problem is np-complete. *Journal of Algorithms*, 10(4):568–576, 1989.

[210] Tommy Thompson. "what is a super mario level anyway?"an argument for non-formalist level generation in super mario bros. 2016.

[211] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.

[212] T. Tieleman and G. Hinton. RMSprop Gradient Optimization.

[213] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-Based Procedural Content Generation. In *Applications of Evolutionary Computation*, pages 141–150. Springer, 2010.

[214] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron

Browne. Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):172–186, 2011.

[215] Christopher W Totten. *An architectural approach to level design.* CRC Press, 2014.

[216] J. Valls-Vargas, S. Ontañón, and Jichen Zhu. Towards story-based content generation: From plot-points to maps. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8, Aug 2013.

[217] Valtchan Valtchanov and Joseph Alexander Brown. Evolving dungeon crawler levels with relative placement. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, C3S2E '12, pages 27–35, New York, NY, USA, 2012. ACM.

[218] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Intl. Conf. on Machine Learning*, pages 1096–1103. ACM, 2008.

[219] David Wagner. A generalized birthday problem. In *Annual International Cryptology Conference*, pages 288–304. Springer, 2002.

[220] Kenneth Wallis. A note on the calculation of entropy from histograms. *http://mpra.ub.uni-muenchen.de/52856/*, 2006.

[221] Noah Wardrip-Fruin. Playable media and textual instruments. *The Aesthetics*

of Net Literature: Writing, Reading and Playing in Programmable Media, pages 211–388.

[222] Max Wertheimer. Experimental studies on seeing motion. *On perceived motion and figural organization*, pages 1–92, 2012.

[223] Yuhong Yang. Can the strengths of aic and bic be shared? a conflict between model indentification and regression estimation. *Biometrika*, 92(4):937–950, 2005.

[224] Georgios N. Yannakakis and Julian Togelius. Experience-Driven Procedural Content Generation. *IEEE Transactions on Affective Computing*, 2(3):147–161, 2011.

[225] R. M. Yerkes and J. D. Dodson. The relation of strength of stimulus to rapidity of habit formation. *Journal of Comparative Neurology and Psychology*, 18:459–482, 1908.

[226] Weidong Yin, Yanwei Fu, Leonid Sigal, and Xiangyang Xue. Semi-latent gan: Learning to generate and modify facial images from attributes. *arXiv preprint arXiv:1704.02166*, 2017.

[227] Andrew Yoder. *Methods of Analysis and Mario*, (accessed April 30, 2018). `https://mclogeblog.wordpress.com/2013/02/24/methods-of-analysis-and-mario/`.

[228] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[229] Tasos Zembylas. *Artistic Practices: Social Interactions and Cultural Dynamics*. Routledge, 2014.

[230] Xiangxin Zhu and Deva Ramanan. Face detection, pose estimation, and landmark localization in the wild. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 2879–2886. IEEE, 2012.

[231] Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. Recurrent highway networks. *arXiv preprint arXiv:1607.03474*, 2016.

[232] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.